An Optimal Linear-Time Algorithm for Interprocedural Register Allocation in High Level Synthesis using SSA Form

Philip Brisk, Member, IEEE, Ajay K. Verma, Paolo Ienne, Member, IEEE

Abstract—An optimal linear time algorithm for interprocedural register allocation in high-level synthesis is presented. Historically, register allocation has been modeled as a graph coloring problem, which is NP-complete in general; however, converting each procedure to Static Single Assignment (SSA) Form ensures a chordal interference graph, which can be colored in \(O(|V|+|E|)\) time; the interprocedural interference graph (IIG) is not guaranteed to be chordal after this transformation. An extension to SSA Form is introduced which ensures that the IIG is chordal, and the conversion process does not increase its chromatic number. The resulting IIG can then be colored in linear time.

Index Terms—High Level Synthesis, (Interprocedural) Register Allocation, Graph Coloring, Static Single Assignment (SSA) Form, Chordal Graph.

I. INTRODUCTION

Register allocation is a fundamental problem in high-level synthesis: the goal is to instantiate the minimum number of registers necessary to store all scalar variables in an application. A similar problem also exists in compilers, where the set of registers is fixed, and the compilers must determine whether to allocate each variable to a register, or to spill it to memory. Both of these problems have been modeled using variations of the graph coloring problem, which is NP-complete in the general case [1-3].

The Static Single Assignment (SSA) Form was introduced in a seminal paper by Cytron et al. [4] in 1990; in the years that followed, SSA Form has emerged as the de-facto program representation for use in optimizing compilers. SSA languages, which predate SSA Form, also have a long and rich history of use in high-level synthesis systems. SSA languages impose strong restrictions on the situations in which programmers could assign values to variables and read their values.

Although somewhat cumbersome for the programmer, these restrictions ensure favorable properties which can be exploited by either a compiler or a high-level synthesis system [5]. The power of the SSA Form is that any sequential program written in a non-SSA language can be converted, automatically, to a representation that has the same restrictions and therefore achieves the same favorable properties as an SSA language; the programmer, however, remains oblivious, and is no longer required to write the application in a restricted SSA language.

Springer and Thomas [5] proved that the interference graph for a program written in an SSA language provably belongs to the class of chordal graphs, which can be colored optimally in \(O(|V| + |E|)\) time, using an algorithm introduced by Gavril [6] in 1972. Several researchers have since rediscovered this result in the equivalent context of SSA Form [7-9].

These theoretical results have resulted in a renewed interest in the use of SSA Form for register allocation, both in compilers and high-level synthesis. There is also interest in the development and exploitation of extensions to SSA Form, whose interference graphs provably belong to known subclasses of chordal graphs. To date, two such extensions have been characterized: the Static Single Information (SSI) Form [10, 11], whose interference graphs are interval graphs [12]; and the Elementary Form [13], whose interference graphs, elementary graphs, are a subclass of interval graphs.

This paper extends these results into the interprocedural domain. It introduces techniques to transform programs whose procedures are converted to SSA, SSI, or Elementary Form to ensure that the interprocedural interference graph (IIG) is a chordal, interval, or elementary graph respectively. This paper extends results previously published at ICCAD 2007, which only account for SSA Form and chordal IIGs [14].

In high-level synthesis, we introduce the first optimal polynomial-time algorithm for interprocedural register allocation. Each procedure is converted to SSA Form; further transformations, which are introduced in this paper, ensure that the IIG is chordal and can be colored optimally in \(O(|V| + |E|)\) time. To date, two prior heuristics have been published for interprocedural register allocation [15, 16], but they do not use SSA Form and they color the IIG using heuristics that offer no guarantee of optimality.

With respect to compilers, Pereira and Palsberg introduced an optimal spill test that can be applied to individual procedures converted to Elementary Form; we describe the
spill test in greater detail in Section II.B. This paper proves that their spill test can be applied interprocedurally without sacrificing optimality. Specifically, we convert each procedure to Elementary Form, and then apply further transformations to ensure that the IIG is an elementary graph; the correctness and optimality of their spill test follows as an immediate corollary.

To date, we are unaware of any problems in register allocation that are NP-complete for SSA Form and chordal graphs, but become polynomial for SSI Form and interval graphs; our interest, here, is purely theoretical. In principle, we believe that it should be possible to develop an SSA-based program representation whose interference graphs provably belong to any class C of graphs that is a superclass of elementary graphs and a subclass of chordal graphs. Using the techniques outlined in this paper, it should be possible to ensure that the IIG belongs to class C as well. SSI Form and interval graphs are presented as one such example.

II. RELATED WORK

A. Register Allocation in High-Level Synthesis

In high-level synthesis, register allocation is typically performed following resource allocation and scheduling, and prior to binding. This last step includes binding of arithmetic operations onto computational resources and variables onto registers, with the goal of minimizing the overall cost of interconnect resources (e.g., wires or busses) and the necessary multiplexers placed on the inputs of the various resources that have been allocated. Often, register binding is performed in conjunction with the other binding phases, and is decoupled from the process by which physical registers are allocated.

Tseng and Siewiorek [3] modeled register allocation for data flow graphs (DFGs) as a problem akin to graph coloring (specifically: clique partitioning on the complement of the interference graph). Kurda and Parker [17] proved that the interference graph, in this context, is an interval graph, and could be colored optimally in polynomial time. Stok [18] proved that the interference graph for a loop is a cyclic interval graph, for which coloring is NP-complete; conversion to SSA Form would break dependencies along the back edges, and the resulting interference graph would be an interval graph.

Springer and Thomas [5] proved that the interference graph for a procedure written in an SSA language is chordal. This result was rediscovered in the equivalent context of SSA Form [7-9]. Springer and Thomas also proved that the IIG is chordal if restrictions are placed on the use of procedure calls.

Vemuri et al. [15] and Beidas and Zhu [16] studied interprocedural register allocation for high-level synthesis. Vemuri et al. built an IIG for the whole program and colored it with a heuristic. Beidas and Zhu developed two scalable heuristics that color each procedure individually and combine the results to form a solution for the whole program. Brisk et al. [14] proved that Beidas and Zhu’s top-down heuristic becomes optimal if the program is converted to a specific SSA-based representation which ensures that the IIG is a chordal graph. This paper also extends Brisk et al.’s work to provide similar extensions for SSI and Elementary Form.

B. Register Allocation in Compilers

In compilers, register allocation can be divided into three distinct sub-problems: the spill test, spilling (sometimes called “allocation”) and coalescing (sometimes called “register assignment”). All three of these sub-problems can be modeled as problems that are similar in principle to graph coloring [1, 2]. The spill test determines whether all of the variables can be stored in registers; if the spill test fails, then spilling partitions the variables that are live at each point in the program between registers and memory. For variables that reside in registers, coalescing assigns variables to registers to eliminate as many copy operations as possible. Spilling [19, 20] and coalescing [21] are NP-complete, even under SSA-based representations.

If the target architecture has $k$ registers, then the spill test is equivalent to testing the interference graph for $k$-colorability [1, 2]. Although NP-complete in the general case, the spill test becomes polynomial under SSA Form where the interference graph is chordal [7-9]; however, this formulation breaks down in the presence of precolored variables and register aliasing, which are features of many modern processors.

For example, many processors contain instructions that read and/or write data from/to specific registers. Precolored variables are introduced to the interference graph to model this situation. Unfortunately $k$-colorability with precolored vertices is NP-complete for interval [22] and unit interval [23] graphs.

Aliasing occurs when registers in the processor’s instruction physically overlap one another; e.g., a 32-bit register may be decomposed into two 16-bit registers. Classic graph coloring cannot account for this [24, 25], as two 16-bit variables whose lifetimes overlap can now be stored in the same 32-bit register.

Periera and Palsberg [13] introduced an optimal polynomial-time spill test that accounts for precolored registers and common cases of register aliasing, for procedures Elementary Form; we extend their result into the interprocedural domain.

Interprocedural register allocation [26, 27] solves the same three problems on the granularity of a whole program. Kurlander and Fischer [27], for example, allocate and assign variables to registers individually for each procedure. They then build an IIG and perform further spilling they achieve $k$-colorability. The interference graph for each procedure is a clique, as registers have been assigned; as a consequence, the IIG is a comparability graph, which can be colored optimally in polynomial time. Even if spilling and coalescing are solved optimally for each individual procedure, the interprocedural solution has no guarantee of optimality.

III. PRELIMINARIES

A. The Control Flow Graph (CFG)

In a compiler’s representation of a procedure, a basic block is a maximum-length sequence of operations with no branches or branch-targets interleaved. Initially, each basic block is a DFG, which becomes linear after scheduling. Early high-level
synthesis systems were limited to DFGs [3, 22], which featured minimal control flow—namely if-then-else and switch statements—via if-conversion [28]. The work that followed, such as that of Springer and Thomas [5], has tried to extend high-level synthesis to procedures, and even whole programs.

Each procedure is represented by a control flow graph (CFG), a directed graph \( G_{\text{CFG}} = (N, E_{\text{CFG}}, \text{entry}, \text{exit}) \); \( N \) is the set of basic blocks, and \( E_{\text{CFG}} \) is the set of control flow edges, i.e., an edge \( (n_i, n_j) \in E_{\text{CFG}} \) indicates that control may transfer from \( n_i \) to \( n_j \); and entry and exit are distinguished basic blocks, both of which are empty; for each basic block \( n \in N_{\text{CFG}} \), there exists at least one path in the CFG from entry to \( n \), and from \( n \) to exit; thus, the CFG is a connected graph.

B. The Call-Points Graph (CPG)

A whole program is comprised of a set of procedures, many of which can call one another. A call point is a location in the program where one procedure calls another. Languages such as C/C++ permit function pointers, which allow a call point to invoke multiple procedures; each time the call point is reached, only one procedure is called, however, the exact procedure called at each invocation can only be determined dynamically.

Unfortunately, pointer analysis is intractable, i.e., it is equivalent to the Halting Problem. Thus, a compiler cannot resolve the precise set of functions that can be called via any given function pointer in the general case. When this occurs, interprocedural analysis and optimization becomes impossible. The methods outlined in this paper—as well as prior work on interprocedural register allocation in compilers [26, 27] and high-level synthesis [5, 15, 16]—are limited to programs for which function pointers can be resolved statically.

The whole program representation that we use is called the call-points graph (CPG). Let \( P \) be the set of procedures in the program, where \( P \in P \) is the entry/exit point (e.g., “main” in C/C++). Let \( C \) be the set of call points in the program, and let \( c_k \in C \) be a point where procedure \( P_k \) calls another procedure \( P_i \) directly, or a function pointer calls a set of procedures \( Q_j \).

The CPG is a directed hypergraph \( G_{\text{CPG}} = (V_{\text{CPG}}, E_{\text{CPG}}) \), where \( V_{\text{CPG}} = P \cup C \); for each call point \( c_k \) where procedure \( P_i \) calls procedure \( P_j \) directly, we add edges \((P_i, c_k)\) and \((c_k, P_j)\) to \( E_{\text{CPG}} \), and for each call point \( c_k \) where procedure \( P_i \) calls a set of procedures \( Q_j \) through a function pointer, we add an edge \((P_i, c_k)\) to \( E_{\text{CPG}} \), along with a hyper-edge \((c_k, Q_j)\) which points to each procedure \( P_i \in Q_j \). Fig. 1 shows an example CPG that will be used to illustrate an important step of our algorithm in the following subsections.

In principle, each hyper-edge \((c_k, Q_j)\) could be replaced with a set of edges, \([(c_k, P_i)]_{P_i \in Q_j}\) without affecting the semantics of the data structure; we prefer to use hyper-edges as doing so eliminates redundant storage of the call point \( c_k \).

C. Liveness and the Interference Graph

Let \( v \) be a variable declared in a procedure. An operation \( v \leftarrow \ldots \) is called a definition of \( v \), and an operation \( \ldots \leftarrow v \) is called a use of \( v \).

Let \( p \) be a point in a procedure; \( v \) is live at \( p \) if there is a path in the CFG from a definition of \( v \) to \( p \) and a path from \( p \) to a use of \( v \). The set of all points where \( v \) is live is called the live range of \( v \). Two variables interfere if their live ranges intersect. An interference graph is for a procedure \( P_i \), an undirected graph \( G_i = (V_i, E_i) \) where \( V_i \) is the set of variables declared locally in \( P_i \), and \( E_i \) contains an edge \((v_i, v_j)\) for every pair of interfering variables.

The interference graph, as defined above, is only applicable to an individual procedure. The interprocedural interference graph (IIG) is defined for a whole program, and includes interprocedural interferences, i.e., interferences between variables that are defined locally in distinct procedures.

If variable \( v \) defined in procedure \( P_i \) is live across a call point \( c_k \), then \( v \) will interfere with every variable defined locally in every procedure reachable from \( c_k \). Referring back to Fig. 1, if \( v \) is defined in \( P_1 \) and is live across call point \( c_9 \), then it will interfere with all variables declared in procedures \( P_3 \) and \( P_5 \) as well; \( v \) will not, for example, interfere with any variables declared locally in procedure \( P_2 \), unless \( v \) is live across call point \( c_7 \) or \( c_8 \) in addition to \( c_9 \).

Two types of variables exist whose lifetimes are implicitly interprocedural. Global variables are defined in a separate global scope that, in principle, coexists with every procedure. Static local variables are declared locally in a procedure; the first time the procedure is called, the value of the static local variable is initialized, and it retains its value across multiple invocations of the procedure. In principle, it should be possible to analyze the precise lifetimes of global and static local variables using an interprocedural extension of the concept of liveness; however, we are unaware of any such technique for doing so that has been published to date.

Our implementation currently assumes that global and static local variables are stored in memory locations; in principle, we could allocate an extra register for each of these variables instead. The development of techniques to precisely analyze the lifetimes of these variables is left open for future work.

The IIG is an undirected graph \( G = (V, E) \), where \( V \) contains the set of all variables declared locally in each procedure, i.e., \( V = V_1 \cup V_2 \cup \ldots \cup V_{\text{CFG}} \), and \( E \) contains all global and local interferences, i.e., \( E = E_1 \cup E_2 \cup \ldots \cup E_{\text{CFG}} \cup E_{\text{global}} \), where \( E_{\text{global}} \) is the set of global interferences. The IIG is used for interprocedural register allocation in both high-level synthesis and compilers.
D. Handling Recursion

Based on the two preceding sections, it is unclear how to handle recursion. Consider, for example, a call point \( c_k \) at which procedure \( P_i \) calls itself. Based on the definition of interprocedural interference above, any variable \( v \) declared locally in \( P_i \) that is live across \( c_k \) interferes with itself, which is an inherent contradiction. In compilers, this conundrum is avoided by pushing \( v \) onto the recursive stack before the call, and popping it off the stack afterwards; this prevents the values of \( v \) instantiated by different invocations of \( P_i \) from interfering with one another. This works because compilers implicitly assume that the number of memory locations is infinite, whereas, the number of registers available is finite.

In principle, if the depth of the recursion can be resolved statically, then the sequence of recursive calls can be unrolled. That is, if the depth is \( d \), we can instantiate \( d \) instances of \( P_i \), denoted \( P_i(1), \ldots, P_i(d) \) in the CPG; this would create \( d \) instances of \( v, v(1), \ldots, v(d) \), all of which interfere with one another interprocedurally.

In the general case, a cycle in the CPG is indicative of recursion; unrolling a fixed-depth recursive call sequence eliminates the cycle. For cycles that cannot be eliminated in this fashion, we compute the strongly connected components (SCC) of the CPG, and collapse all of the vertices (including procedures and call points) into a single vertex. All variables that are live across a call point that has been collapsed in this fashion are pushed onto a stack in memory, as suggested above, eliminating all recursive interprocedural interferences.

This does require a small amount of bookkeeping. Let \( S \) denote a cycle in the CPG, and consider a call point \( c_k \) where procedure \( P_i \in S \) calls procedure \( P_j \notin S \). After collapsing all of the vertices in \( S \) into a single vertex, the CPG contains edges \((S, c_k)\) and \((c_k, P_j)\). The former edge, \((S, c_k)\), is augmented to account for the fact that the caller, in actuality, is \( P_i \in S \).

Collapsing each SCC into a single vertex eliminates all cycles in the CPG; to simplify further discussion, we assume that the CPG is acyclic, and that each vertex in the CPG contains a single procedure or call-point. Extensions to account for recursive calls are straightforward, but require a fair amount of bookkeeping in order to implement properly.

IV. SCALABLE INTERPROCEDURAL REGISTER ALLOCATION

In this section, we briefly summarize Beidais and Zhu’s [16] scalable heuristics for interprocedural register allocation; they introduced two heuristics, top-down, and bottom-up color palette propagation, which are reviewed in the following subsections. These heuristics are general frameworks that color the interference graph for each procedure individually and propagate coloring constraints; the frameworks do not depend on the specific methods used for coloring.

A. Enumeration of Interprocedural Interferences

In the general case, there may be many paths in the CPG from the entry procedure \( P_1 \) to a given procedure \( P_i \). For any specific invocation of \( P_i \) the calling context is the specific path taken which leads to the call.

Context-sensitive compiler analyses, for example, may choose to clone \( P_i \) so that different optimizations can be applied depending on its calling context [29]. Register allocation, in this case, would be applied afterwards, so the different instances of \( P_i \) are treated as distinct procedures.

For interprocedural register allocation, it is useful to know the maximum number of interprocedural interferences that occur among all possible paths in the CPG from \( P_i \) to \( P_j \); we denote this quantity by \( \delta_i \). In principle, this information can be gleaned through exhaustive enumeration of all paths in the CPG; however, a more efficient algorithm to compute this information also exists, due to the fact that the CPG is acyclic.

Suppose that procedure \( P_i \) calls procedure \( P_j \) at call point \( c_k \). Let \( L(c_k) \) denote the number of variables defined locally in \( P_i \) that are live across \( c_k \). Given \( \delta_i \), it is straightforward to see that \( \delta_i \geq \delta_j + |L(c_k)| \), as there is no guarantee that this path contains the maximum number of interprocedural interferences leading into \( P_i \). Since the CPG is acyclic, it is straightforward to see that we can process all of the procedures that call \( P_i \) before considering \( P_i \) by processing the CPG vertices in topological order. To simplify the discussion, we also associate a quantity \( \delta \) with each call point. Specifically, if \( c_k \) occurs in procedure \( P_n \), then \( \delta_k = \delta_j + |L(c_k)| \).

Inductively, the process works as follows. \( \delta_i \), corresponding to the entry procedure, is a constant. In principle, \( \delta_i \) may be zero, however, languages such as C/C++ pass two parameters, arge and argy to main, so in this case, \( \delta_i = 2 \). The specific constant used for \( \delta_i \) depends on the language, operating system, and or programmer decisions involved. This establishes the basis. For the induction hypothesis, assume that \( \delta_i \) is known for each CPG vertex \( x \) such that the maximum distance from \( P_i \) to \( x \) in the CPG is at most \( t \) hops. Now, consider a vertex \( y \in V_{CPG} \) such that the maximum distance from \( P_i \) to \( y \) is \( t+1 \) hops. There are two cases to consider:

1. If \( y \) is a call point, \( c_k \), then the maximum distance from \( P_i \) to the caller \( P_j \) is at most \( t \) hops, so \( \delta_j \) is known. As \( |L(c_k)| \) is a known constant, given that variable lifetimes in \( P_i \) have been analyzed in advanced, it follows that \( \delta_k = \delta_j + |L(c_k)| \).

2. If \( y \) is a procedure, \( P_j \), let \( C_j \) be the set of points that call \( P_j \). As the CPG is acyclic, it follows that each call point \( c_k \in C_j \) is at most \( t \) hops from \( P_i \), so \( \delta_j \) is known. As every path from \( P_i \) to \( P_j \) goes through precisely one call point in \( C_j \), it follows that

\[
\delta_k = \max_{c_k \in C_j} \{ \delta_j \}. \tag{1}
\]

Therefore, it follows that \( \delta_k \), corresponding to procedure \( P_j \), is the maximum number of global interferences on any path from \( P_i \) to \( P_j \). Table I shows an example, corresponding to the CPG in Fig. 1; in this case, we assume that \( \delta_j = 0 \), and values of \( |L(c_k)| \) for each call-point are given. Processing the vertices in the CPG in Fig. 1 in topological order yields the \( \delta \) values for each procedure and call point, as shown in Table I.
B. Top-Down Color Palette Propagation

In this method, the vertices in the CPG are processed in topological order. Consider, now, a procedure $P_i$ that calls procedure $P_j$ at call point $c_i$. Recall that $L(c_i)$ is the set of variables that are live across $c_i$, and let $\text{colors}[L(c_i)]$ denote the set of colors that are assigned to the variables in $L(c_i)$. After coloring $P_i$’s interference graph, the heuristic propagates the palette of available colors down from CPG from $P_i$ to $P_j$; specifically, this propagation ensures that no variable defined locally in $P_j$, or any procedure reachable from $P_j$ can receive any of these colors. The set of unavailable colors is propagated down from all of the call points that call $P_j$, which is feasible due to the fact that the CPG is acyclic and its vertices are processed in topological order. For each call point $c_i$ in $P_j$, the union of $\text{colors}[L(c_i)]$ and $\text{colors}[L(c_i)]$ is propagated to the procedure called from $c_i$.

If we assume that each procedure is converted to SSA Form, and has a chordal interference graph, it is not immediately clear why this heuristic is sub-optimal. In fact, the algorithm is optimal if the CPG is a tree, i.e., each procedure is only called from one call point.

Recall that $G_i$ is the interference graph for procedure $P_i$, and let $\chi_i$ denote the chromatic number of $G_i$. In this restricted case, it is straightforward to see that exactly $\delta_i$ colors will be unavailable on the palette when $G_i$ is colored; hence, the total spectrum of colors used to color $G_i$, and all variables live across calls leading to $G_i$, will be $\delta_i + \chi_i$. In this case, the chromatic number of the IIG $G$, denoted $\chi(G)$, is

$$
\chi(G) = \max_{P \in P} \{\delta_i + \chi_i\}.
$$

Optimality is lost, however, when multiple distinct call points call the same procedure. For example, suppose that procedure $P_1$ calls procedure $P_2$ twice, at call points $c_3$ and $c_5$. For the sake of simplicity, assume that $L(c_3) = \{v_1\}$ and $L(c_5) = \{v_2\}$, and that $\text{color}(v_1) = 1$ and $\text{color}(v_2) = 2$. In this case, $\delta_i = 1$, but two colors are unavailable on the palette, so the spectrum of colors used to color $G_i$ will be $\{3, 4, ..., \chi_5 + 2\}$. In principle, optimality could be guaranteed if, for every call point $c_i$ that calls $P_2$, we can ensure that the palette of unavailable colors is $\{1, 2, ..., \delta_i\}$; however, this constrains the colors that can be assigned to variables defined locally in each procedure, depending on which call points each variable is live across. This problem is effectively the same as coalescing in compilers, as we impose additional constraints that require that various pairs of non-interfering vertices in the interference graph receive the same color. As coalescing is NP-hard for chordal graphs [19], top-down color palette propagation is NP-hard as well for programs where each procedure has been converted to SSA Form.

The primary contribution of this paper is an interprocedural extension to SSA Form that allows us to sidestep this conundrum. Our method effectively allocates a set of registers whose primary purpose is to hold variables that are live across procedure calls; specifically, we ensure that the variables live across calls in a path leading to procedure $P_j$ are assigned to registers corresponding to colors $\{1, 2, ..., \delta_j\}$. Details are provided in Section VI.

C. Bottom-Up Color Palette Propagation

Bottom-up color palette propagation processes procedures in reverse topological order. In other words, the interference graph of procedure $P_i$ is only colored after every procedure reachable from $P_i$ in the CPG is colored first. For example, if procedure $P_i$ calls procedure $P_j$ at call point $c_i$, then the colors assigned to variables in $P_j$, and every procedure reachable from $P_j$, are available for variables in $L(c_i)$.

To see why this approach remains NP-complete, even when all procedures are converted to SSA Form, consider a simple program where procedure $P_1$ calls procedures $P_2$ and $P_3$ at call points $c_4$ and $c_5$, respectively. First, we color $P_2$ and $P_3$ optimally, using colors $\{1, 2, ..., \chi_2\}$ and $\{1, 2, ..., \chi_3\}$ respectively. To propagate these colors upward, we instantiate $\max(\chi_2, \chi_3)$ pre-colored vertices in $G_i$, the interference graph for $P_1$. Each vertex belonging to $L(c_i)$ and $L(c_j)$ is interferes with the first $\chi_2$ and $\chi_3$ pre-colored vertices respectively. Although chordal graph coloring has an efficient polynomial-time algorithm in the general case, the problem becomes NP-complete in the presence of pre-colored vertices for interval [22] and unit-interval [23] graphs, which are subclasses of chordal graphs. Thus, the process of propagating coloring constraints upwards through the CPG renders interprocedural register allocation NP-complete for procedures in SSA Form whose interference graphs are chordal.

V. PARALLEL COPY OPERATIONS

All three SSA-based representations considered in this paper split the live ranges of variables by introducing specific types of parallel copy operations. The conversion to SSA Form introduces $\varphi$-functions, which are placed at the beginning of basic blocks having multiple predecessors. In addition to $\varphi$-functions, SSI Form requires a second type of parallel copy, called a $\pi$-function, which are placed at the end of basic blocks having multiple successors. In addition to $\varphi$- and $\pi$-functions, Elementary Form uses parallel copy operations within basic blocks to split all of the variables that are live at each and every point in the program.
A. \(\phi\)-functions

Fig. 2 illustrates \(\phi\)-functions. In Fig. 2(a), variable \(v\) is defined twice on two different sides of a condition, and is used afterwards. The two definitions of \(v\) are renamed to \(v_1\) and \(v_2\), as shown in Fig. 2(b); however, renaming the use of \(v\) to either \(v_1\) or \(v_2\) would alter the semantics of the program. To rectify the situation, a \(\phi\)-function is placed at the entry point of the basic block following the two sides of the condition. The \(\phi\)-function, merges the values of \(v_1\) and \(v_2\), depending on which side of the condition is taken, and defines a new variable \(v_3\); the use of \(v\) is then renamed to be a use of \(v_3\) instead, preserving the correct semantics of the program.

In the general case, a basic block may contain multiple \(\phi\)-functions instantiated for several variables at its entry point. A set of \(\phi\)-functions can be represented as a matrix [8]:

\[
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_k
\end{bmatrix} = \Phi 
\begin{bmatrix}
x_{11} & x_{12} & \cdots & x_{1m} \\
x_{21} & x_{22} & \cdots & x_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
x_{k1} & x_{k2} & \cdots & x_{km}
\end{bmatrix}.
\]

(3)

If control enters a basic block from its \(i^{th}\) predecessor, then a parallel copy \([y_{1i}, y_{2i}, \ldots, y_{ki}] \leftarrow pcopy[x_{1i}, x_{2i}, \ldots, x_{ki}]\) executes. Variables \(y_{1i}, y_{2i}, \ldots, y_{ki}\) interfere with one another, as do \(x_{1i}, x_{2i}, \ldots, x_{ki}\); however, the parallel copy prevents any the \(y_{ji}\)’s from interfering with the \(x_{ji}\)’s. Moreover, no \(x_{ji}\) variables in different columns interfere on account of the \(\phi\)-function, but they may interfere if their lifetimes intersect elsewhere in the program.

B. \(\pi\)-functions

Fig. 3 illustrates \(\pi\)-functions. In Fig. 3(a), variable \(v\) is defined prior to a condition, and is used on both sides of the condition. In Fig. 3(b), the definition of \(v\) is renamed to \(v_1\); a \(\pi\)-function is inserted to split \(v_1\) into two new variables, \(v_2\) and \(v_3\), each of which is used on a different side of the condition. The uses of \(v\) on the two sides are renamed to \(v_2\) and \(v_3\) respectively, preserving the semantics of the original program.

Like \(\phi\)-functions, a set of \(\pi\)-functions at the end of a basic block can be represented using matrix notation:

\[
\begin{bmatrix}
y_{11} & y_{12} & \cdots & y_{1m} \\
y_{21} & y_{22} & \cdots & y_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
y_{k1} & y_{k2} & \cdots & y_{km}
\end{bmatrix} = \Pi 
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_k
\end{bmatrix}
\]

(4)

If control flows from \(n\) to its \(i^{th}\) successor, then a parallel copy operation \([y_{j1}, y_{j2}, \ldots, y_{ji}] \leftarrow pcopy[x_1, x_2, \ldots, x_i]\) executed. Variables \(x_1, x_2, \ldots, x_i\) all interfere with one another, as do \(y_{j1}, y_{j2}, \ldots, y_{ji}\); however, the parallel copy prevents any of the \(y_{ji}\) from interfering with any of the \(x_i\) variables. Moreover, no two \(y_{ji}\) variables in different columns of the matrix interfere on account of the \(\pi\)-function, but they may interfere if their lifetimes intersect elsewhere in the program.

C. Parallel Copies Inside Basic Blocks

Both \(\phi\) and \(\pi\)-functions are parallel copy operations; \(\phi\)-functions are effectively placed on the CFG edges that come into a basic block, while \(\pi\)-functions are effectively placed on outgoing edges. The Elementary Form [13] requires a third type of parallel copy operation in addition to \(\phi\) and \(\pi\)-functions. These parallel copies do not have special names or notation, and may be placed between any pair of scheduled operations in a basic block. We let \(p\) denote the point in the program, and \(L(p)\) denote the set of variables live at \(p\). The parallel copy defines a new set of variables, denoted \(L'(p)\), and is denoted by \(L'(p) \leftarrow pcopy[L(p)]\). Each use of a variable \(v \in L(p)\) that is reachable from \(p\) is replaced by a use of \(v'\), the variable corresponding to \(v\) in \(L'(p)\).

Parallel copy operations and \(\phi\) and \(\pi\)-functions are virtual entities. A compiler must eventually serialize them into non-parallel copy operations, or a high-level synthesis tool must allocate the necessary wires and control signals between registers to implement these operations.

VI. SSA-BASED PROGRAM REPRESENTATIONS AND THEIR INTERFERENCE GRAPHS

This section introduces three SSA-based representations which are used for individual procedures, and their corresponding interference graph classes. Here, we only consider pruned SSA-based representations [30]. Pruned SSA, SSI, and
Elementary Form only insert \( \varphi \)- and \( \pi \)-functions for a variable \( v \) at points where \( v \) is live. These pruned representations partition each variable into a set of variables with smaller lifetimes. Suppose that \( v \) is replaced with a collection of new variables, \( v_1, v_2, \ldots, v_k \); then: (1) the union of the live ranges of \( v_1, v_2, \ldots, v_k \) is precisely the live range of \( v \); and (2) the live ranges of any two distinct variables \( v_i \) and \( v_j \) do not intersect. Therefore, the conversion to any of these SSA-based program representations cannot increase the chromatic number of the interference graph; prior work has shown that, in certain cases, the conversion to SSA Form can actually reduce the chromatic number of the interference graph [7-9].

The precise definitions of SSA, SSI, and Elementary Form are quite detailed and are not actually needed in order to understand this paper; all that is required is to understand the corresponding interference graphs classes. They do, however, share one common property: each variable is renamed so that each variable is defined once, and each use of a renamed variable corresponds perfectly to its one definition.

Fig. 4 shows a program fragment converted to SSA and SSI Form; Fig. 5 shows the fragment converted to Elementary Form. Elementary Form, it should be noted, inserts the maximum possible number of non-redundant parallel copies, and entails significantly more \( \varphi \)- and \( \pi \)-functions than SSA or SSI Form. Specifically, a \( \varphi \)-function for variable \( v \) is placed at the entry point of every basic block having more than two predecessors where \( v \) is live; a \( \pi \)-function for \( v \) is placed at the exit point of every basic block having multiple successors where \( v \) is live; and a parallel copy operation that splits \( v \) (and possibly other variables as well) is placed between pair of scheduled operations inside a basic block where \( v \) is live.

A. Chordal Graphs
Let \( G = (V, E) \) be an undirected graph. A \( k \)-cycle is a sequence of vertices \( <v_0, v_1, \ldots, v_k, v_0> \) where \((v_i, v_{(i+1) \mod k}) \in E, 0 \leq i \leq k-1\). A chord is an edge \( (v_i, v_j) \) between two non-adjacent vertices in the cycle, i.e., \( j \neq i+1 \mod k \) and \( i \neq j+1 \mod k \). Fig. 6 shows an example of a 4-cycle with a chord.

A chordless cycle of length at least four is called a hole. A chordal graph is a hole-free undirected graph. Other provably equivalent definitions exist, but will not be used in this paper. For our purposes, it suffices to note that the interference graph for a procedure in SSA Form is a chordal graph [7-9], and chordal graphs can be colored optimally in \( O(|V| + |E|) \)-time [6]. The 5-hole is a noteworthy graph for the following reason. It is the smallest graph whose chromatic number, 3, is larger than the size of its largest clique, 2; in fact, any \( k \)-hole, where \( k \) is odd, retains this property. As chordal graphs contain no holes, the chromatic number is always equal to the cardinality of the largest clique. The conversion to SSA Form, therefore, can reduce the chromatic number of the interference graph by eliminating odd-length holes [7, 8].

B. Interval Graphs
An interval graph is the intersection graph between a set of intervals on the real number line. In other words, we can associate an interval, \([low, high]\), with each variable \( v \), and an edge is placed between two variables if and only if their intervals intersect. For our purposes, it suffices to note that the interference graph for a procedure in SSI Form is an interval graph, and that interval graphs are a subclass of chordal graphs, which can be colored optimally in \( O(|V|) \)-time [31].

---

Fig. 4. A program fragment represented as a CFG (a), and converted to SSA (b) and SSI Form (c). The NULL definition in (c) occurs because variable \( y \) is not live outside of the loop in the CFG fragment in (a).
Fig. 7 shows an instruction sandwiched between two parallel copy operations; the interference graph for this instruction is formed from two cliques, $C_1$ and $C_2$. $C_1 \cap C_2$ is the set of variables that are live before and after the instruction. Without loss of generality, $C_1 \cap C_2$ is the set of variables that are live before the instruction (defined by the first copy), but are no longer live after the instruction, i.e., the instruction is their final use; similarly, $C_2 \cap C_1$ is the set of variables that are not live before the instruction, but are live after it—i.e., the variables that are defined by the instruction.

VII. INTERPROCEDURAL EXTENSIONS TO SSA, SSI, AND ELEMENTARY FORM, AND IIG PROPERTIES

Consider procedure $P_i$ with interference graph $G_i$. Based on the discussion in Section IV, each variable $v$ declared locally in $P_i$ interferes interprocedurally with at most $\delta_i$ variables in any calling context leading from $P_i$ into $P_i'$, and all variables along any one of these paths interfere with one another as well. As we will shortly see, the IIG for a program where each procedure is represented in SSA Form is not necessarily chordal, due to interprocedural interferences. In addition to the complications outlined in Sections IV.B and IV.C, the IIG itself may contain odd-length holes whose elimination, in principle, could reduce the chromatic number. In this section, we describe how to transform the program, without increasing the chromatic number of the IIG, in a way that eliminates all holes of length four or greater in the IIG.

We provide two answers to this conundrum. The first answer, introduced in subsection A, is to allocate a stack of global registers which holds variables that are live across procedure calls; variables that are live across the call are pushed onto the stack in parallel before the call, and popped from the stack in parallel afterwards. This suffices to ensure that the IIG is a chordal or interval graph if each procedure is respectively converted to SSA or SSI Form; however, it cannot ensure that the IIG is an elementary graph if each procedure is converted to Elementary form. This is because the semantics of Elementary form inserts parallel copies for all variables—include those residing in the global stack—at each point where they are live.

To rectify the situation, our second proposal is to pass all $\delta_i$ variables that are live across calls leading to procedure $P_i$ as parameters, so that the parallel copies that are inserted to convert $P_i$ to Elementary Form can copy them too; $P_i$ must also pass them back to the caller upon termination.

A. Caller-save Semantics: Launch and Landing Pads

Under this strategy, we allocate a stack of global registers that hold variables that are live across procedure calls. In other words, each variable declared locally in procedure $P_i$ will interfere with exactly $\delta_i$ of these global registers. Alternatively, each variable defined locally in $P_i$ would interfere with $\delta_i$ variables defined locally in other procedures for every unique path in the CPG leading to $P_i$. 

C. Elementary Graphs

Due to the aggressive instantiation of parallel copy operations in an Elementary Form program, each variable lifetime spans at most one instruction: the maximum lifetime of a variable is from the parallel copy that precedes the instruction to the parallel copy that follows it. As a consequence, an elementary graph satisfies the property that each connected component is either a clique, or two intersecting cliques. Fig. 7 shows an example.
Let $\delta_{\text{max}} = \max\{|\delta_i| : 1 \leq i \leq |P|\}$; $\delta_{\text{max}}$ is the maximum number of variables live across call points among every path in the CPG. Therefore, we allocate a set $T = \{t_1, t_2, ..., t_M\}$ of global registers, where $M = |T| = \delta_{\text{max}}$. Now, consider a call point $c_i$ where procedure $P_i$ calls procedure $P_j$. To simplify notation, let $m = \delta_i$ and $n = |L(c_i)|$ be the number of variables live across the call. Inductively, we assume that all of the variables that are live across calls from which $P_i$ is reachable are assigned to global registers $\{t_1, ..., t_m\}$. $P_i$ then copies the variables in $L(c_i)$ to the next subset of available global registers, $\{t_{m+1}, ..., t_{m+n}\}$, effectively pushing them onto a stack; in parallel $P_j$ passes all of the necessary parameters to $P_j$. Next, $P_j$ executes, upon completion, control returns to $P_i$, who pops $n$ variables off of the stack, restoring them to local storage. We refer to these parallel push/pop operations as Launch and Landing pads, denoted by $\Psi$ and $\Psi^\dagger$ respectively.

Thus, a procedure call operates as follows:

$$(t_{m+1}, ..., t_{m+n}) \leftarrow \Psi(L(c_i)) \mid \text{Pass parameters from } P_i \text{ to } P_j$$

$$L'(c_i) \leftarrow \Psi^\dagger(t_{m+1}, ..., t_{m+n})$$

As the landing pad assigns new values to the variables in $L(c_i)$, they are renamed (denoted by the set $L'(c_i)$); this is required because the definition of SSA Form mandates that each variable is only assigned a value once. Each use of a variable in $L(c_i)$ that is reachable from the landing pad is replaced with the corresponding renamed variable in $L'(c_i)$.

Fig. 8 illustrates the insertion of launch and landing pads. Fig. 8(a) shows a program with two procedures, $A$ and $B$, both converted to SSA Form. Fig. 8(b) shows the program after inserting launch and landing pads at the point where $A$ calls $B$.

Figs. 8(c) and (d) show the IIGs corresponding to the programs in Figs. 8(a) and (b). The IIG shown in Fig. 8(c) is a 5-hole, which is non-chordal, and has a chromatic number of 3. The IIG shown in Fig. 8(d) is chordal, and has a chromatic number of 2. Therefore, similar to the conversion to SSA Form in the intraprocedural case [7, 8], the insertion of launch and landing pads can reduce the chromatic number of the IIG.

In principle, any local procedure $P_l$ can bind variables declared locally to any global register $t_i$, $l > \delta$. After the insertion of launch and landing pads, no locally declared variable will be live across procedure calls.

In the case of Fig. 8(c), the chromatic number of the IIG is 2, but there is only one global register, $t_1$. In a minimum coloring, some of the variables defined locally in $A$ must be assigned to $t_1$. The sub-optimal and unnecessary alternative is to allocate a third register to the design.

Next, we prove the desired properties of the IIG. The two lemmas that follow form the building blocks of our proof.

**Lemma 1.** After the insertion of launch and landing pads, no two variables defined locally in distinct procedures interfere.

**Proof.** Let $P_i$ and $P_j$ be procedures, and with local variables $v_i$ and $v_j$ respectively defined in $P_i$ and $P_j$. Without loss of generality, $v_i$ can only interfere with $v_j$ if $v_i$ is live across a call $c_k$ in $P_i$ and there is a path from $c_k$ to $P_j$ in the CPG; however, a launch pad placed before $c_k$ in $P_i$ copies $v_i$ to a global register $t_i \in T$, so $v_i$ and $v_j$ cannot interfere: a contradiction. $\square$

**Lemma 2.** After the insertion of launch and landing pads, $T$ forms a clique in the IIG.

**Proof.** Follows immediately from the fact that $\delta_{\text{max}}$ is the maximum number of variables live across call points among every path in the CPG, and $|T| = \delta_{\text{max}}$. $\square$
Lemmas 1 and 2 generally characterize the IIG of a program after inserting launch and landing pads. As shown in Fig. 9, T forms a clique, and all variables defined locally in $P_i$ interfere with global registers $\{t_{i_1}, ..., t_{i_m}\}$ where $m = \delta_i$. Let $\langle G_i = (V_i, E_i) \mid 1 \leq i \leq N \rangle$ be the set of interference graphs for the procedures comprising the whole program.

We let $G_T = (T, E_T)$ represent the interference graph for the global registers, which is a clique by Lemma 2. By Lemma 1, the IIG, $G = (V, E)$, is constructed as follows:

$$V = T \cup \bigcup_{i=1}^{N} V_i, \quad (5)$$

$$E = E_T \cup \bigcup_{i=1}^{N} \left( E_i \cup \bigcup_{j=1}^{\delta_i} \{t_j, v\} \right). \quad (6)$$

**Theorem 1.** After the insertion of launch and landing pads, the chromatic number of the IIG $G$ is

$$\chi(G) = \max_{P_i \in P} \{\delta_i + \chi_i\}. \quad (7)$$

**Proof.** First, let us allocate the set of $M$ global registers: $T$. By Lemma 2, $T$ is a clique, and is colored with $M$ colors. Now, let us add interference graphs for the individual procedures one-by-one. Consider interference graph $P_i$ with interference graph $G_i$, and let $m = \delta_i$. By the insertion of launch and landing pads and by Lemma 1, each variable $v$ defined locally in $P_i$ interferes with global registers $\{t_{i_1}, ..., t_{i_m}\}$, and no local variables defined in other procedures. Therefore, it suffices to color $G_i$ using colors $\{m+1, ..., m+\chi_i\}$, since $\{1, ..., m\}$ are unavailable. Eq. (7) is satisfied by construction. □

Theorem 1 does not rely on properties of the interference graph $G_i$. If SSA Form is not used, an optimal coloring cannot be found in polynomial time, under the assumption that $P \neq NP$. Additionally, this approach is scalable, similar to the heuristic of Beidas and Zhu [16]. It suffices to color $G_i$, and adding $\chi_i$ to the color assigned to each vertex. This approach is top-down, as the set of unavailable colors, $\{1, ..., \chi_i\}$, are propagated from caller to callee.

**Theorem 2.** If each procedure is converted to SSA Form and launch and landing pads are inserted, then the IIG $G$ is chordal.

**Proof.** Assume to the contrary that $G$ is non-chordal; then $G$ must contain a $k$-hole for some $k \geq 4$. The $k$-hole cannot occur in any $G_i$, which is chordal, or in $T$, which is a clique. The $k$-hole cannot occur in any subgraph of $G$ induced by $G_i \cup T$, since each global register $t \in T$ is either adjacent to every vertex in $V_i$, or no vertices in $V_i$. Therefore, the $k$-hole must include vertices $v \in V_i$ and $v \in V_j$, corresponding to variables declared locally in distinct procedures. The $k$-hole is decomposed into paths $X_1 = (v_1, ..., v_j)$ and $X_2 = (v_j, ..., v_k)$, whose only common vertices are $v_1$ and $v_k$. By Lemma 1, $v_1$ and $v_k$ do not interfere, so $X_1$ and $X_2$ must each contain distinct global registers. Since $T$ is a clique, these global registers interfere, as shown in Fig. 10. This indicates the presence of a chord: a contradiction. □

Without Theorem 1, we cannot achieve an optimal coloring of the IIG in polynomial-time, unless $P = NP$ in the general case. When the IIG becomes chordal, we achieve an optimal polynomial-time solution because we can color the interference graph for each procedure $G_i$ in $O(|V_i| + |E_i|)$ time [6]; moreover, we can color the variables in $V_i$ without explicitly constructing $G_i$ using data structures already present in the compiler [7, 8]. By Theorem 1, this coloring is optimal.

**Theorem 3.** After converting each procedure to SSI Form and inserting launch and landing pads, the IIG is an interval graph.

---

**Fig. 9.** The interprocedural interference graph for the program in Fig. 1 and Table I.

**Fig. 10.** Illustration of the proof of Theorem 2. If we assume that $G$ is non-chordal, then a $k$-hole involving two global registers must be present. The $k$-hole involves paths $X_1$ from $v_1$ to $v_j$ and $X_2$ from $v_j$ back to $v_k$, with no common vertices other than $v_1$ and $v_k$. By Lemma 1, both of these paths must contain at least one global register; the edge between these two global registers is a chord.
Proof. The interference graph $G_i$ for procedure $P_i$ is an interval graph; we construct the interval representation of each variable defined locally in $P_i$ such that two variables interfere if and only if their respective intervals overlap. We then enclose these intervals within a larger interval $I_i = [l_i, h_i]$, which denotes the interval representation of $G_i$ as a whole.

To form the interval representation of the IIG, we place all of the intervals on the real number line. We sort and rename the intervals in increasing order of $\delta$-values, such that $\delta_i \leq \delta_j$ for every pair of consecutive intervals $I_i, I_j$. In accordance with Lemma 1, none of the intervals overlap. Let $\alpha_i$ be any value that satisfies $h_i < \alpha_i < l_{i+1}$ for $1 \leq i \leq N-1$, and let $\alpha_N = h_N + 1$. Global register $t_i$ interferes with every variable defined locally in a procedure $P_i$ having $\delta_i \geq k$. Therefore, we can find an index $j$ such that $\delta_j < k$ and $\delta_j \geq k$; by convention, we let $\delta_i = 0$. As the intervals have been sorted, $t_i$ interferes with every variable defined locally in $\{P_i, \ldots, P_{j-1}\}$ and no variables defined locally in $\{P_{j+1}, \ldots, P_{i-1}\}$. Therefore, we associate the interval $I(t_i) = [\alpha_i, \alpha_j]$ with $t_i$. By construction, $I(t_i)$ subsumes the intervals $[l_i, \ldots, l_j]$ and does not overlap $[l_{i+1}, \ldots, l_j]$. The resulting IIG is an interval graph by construction.

After allocating intervals for all global registers, Lemma 2 is satisfied since all of these intervals overlap at $\alpha_N$. $\square$

Chordal graphs can be colored in $O(|V| + |E|)$ time [6]; interval graphs, in contrast, can be colored in $O(|V|)$ time [31]. The chromatic number of $G_i$ and $G$ will be the same, regardless of whether $P_i$ is converted to SSA or SSI Form. To date, no SSI-based register allocator has been implemented and compared to an SSA-based allocator; however, should SSI-based register allocation come into vogue, Theorem 2 ensures that the overall approach can be extended into the interprocedural domain, while ensuring that the IIG remains an interval graph.

B. Extended Launch and Landing Pads

Unfortunately, Theorems 2 and 3 do not generalize to Elementary Form; Fig. 11 shows an example. Fig. 11(a) shows an elementary interference graph $G_i$ for a procedure $P_i$, presumably converted to Elementary Form.

Assuming that $\delta_i = 1$, we allocate a global register $t_j$ which interferes with every variable in $V$, as shown in Fig. 11(b). Unfortunately, the resulting IIG is no longer an elementary graph. Recall that in an elementary graph, each connected component is comprised of at most two intersecting cliques. Adding global registers to form the IIG connects these components, and the resulting component is no longer the intersection of up to two cliques.

The only way to ensure that the IIG becomes an elementary graph is to allow the callee procedure to modify the global registers. This works because at most $\delta_i$ variables will be stored in global registers, depending on the calling context, and $\delta_i$ can be computed statically. When converting the callee to Elementary Form, each $\varphi$-function, $\pi$-function, and parallel copy operation is extended to account for the global registers.

The global registers must be given a new name every time that they are copied. Here, we will consider the semantics of a procedure call from the perspective of the caller. Consider a call point $c_i$ where $P_i$ calls $P_j$, where $m = \delta_i$ and $n = |L(c_j)|$. At the call point, assume that the names of the global registers are $t_{i(1)}, \ldots, t_{i(m)}$. Then, at any point in the program, all global registers currently in use will have the same superscript.

In the preceding section, the launch pad copied the variables in $L(c_i)$ to global registers $t_{m+i}, \ldots, t_{m+n}$, and passed all parameters from $P_i$ to $P_j$ in parallel. To facilitate Elementary form, the launch pad must be extended with a parallel copy operation that also copies $t_{i(1)}, \ldots, t_{i(m)}$ as well. We refer to this operation as an extended launch pad:

$$
(t_{1(1)}, \ldots, t_{m(1)}) \leftarrow \text{pcopy}(t_{i(1)}, \ldots, t_{i(m)})
$$

$$
| | (t_{m+i(1)}, \ldots, t_{m+n(1)}) \leftarrow \Psi(L(c_j))
$$

$$
| | \text{Pass parameters from } P_i \text{ to } P_j
$$

Next, $P_i$ calls $P_j$. All of the parallel copy operations in $P_j$ will be extended to account for the $m+n$ global registers. From the perspective of $P_i$, $P_j$ is atomic and performs the copy:

$$
(t_{1(2)}, \ldots, t_{m(2)}) \leftarrow \text{pcopy}(t_{i(2)}, \ldots, t_{i(m+1)})
$$

In actuality, $P_j$ is unaware of how many times $P_i$ copies the global registers during its execution. Since consecutive copies are transitive, these details are extraneous.

Lastly, the landing pad restores the variables copied to the global registers back to $L'(c_j)$. The landing pad is augmented with a parallel copy operation on the first $m$ global registers as well; the result is called an extended landing pad:

$$
(t_{i(3)}, \ldots, t_{m(3)}) \leftarrow \text{pcopy}(t_{i(3)}, \ldots, t_{i(m+2)})
$$

$$
| | (L'(c_j)) \leftarrow \Psi'(t_{m+i(3)}, \ldots, t_{m+n(3)})
$$

Theorem 4. If each procedure is converted to Elementary Form and extended launch and landing pads are inserted, then the IIG $G$ is an elementary graph.

Proof. Consider procedure $P_i$, which has been converted to Elementary Form; its interference graph, $G_i = (V_i, E_i)$ is an
elementary graph. In the IIG, each vertex in \( V_i \) will interfere with precisely \( m = \delta_i \) global registers: \( \{t_1, \ldots, t_m\} \). Each parallel copy operation that has been inserted to convert \( P_i \) to Elementary Form is extended to copy the global registers as well. This transformation accounts for all interprocedural interferences involving variables defined locally in \( P_i \). Let \( t_j^{(i)} \) denote the \( k^{th} \) copy of global register \( t_j \), \( 1 \leq j \leq m \), that has been instantiated. By construction, \( t_j^{(i)} \) will be defined by one parallel copy; it will remain live across one instruction, \( I \), that will neither use nor redefine it; and it will be used by a parallel copy operation that follows, redefining it as \( t_j^{(k+1)} \). Recall that in Elementary Form, each connected component in the interference graph is comprised of at most two intersecting cliques, \( C_1 \) and \( C_2 \), where \( C_1 \cap C_2 \) is the set of variables that are live before and after the instruction; hence \( t_j^{(k)} \in C_1 \cap C_2 \) for the connected component corresponding to instruction \( I \).

Now, we consider the process by which procedure \( P_i \) calls \( P_j \) at call point \( c_i^j \); the extended launch and landing pads themselves are simply parallel copy operations; in essence, we can view them as parallel copy operations that surround an actual instruction in the context of Elementary Form; that one instruction is the call to \( P_j \) which executes in parallel with a copy \( \{t_1^{(3)}, \ldots, t_{m+n}^{(3)}\} \leftarrow \text{pcopy}\{t_1^{(2)}, \ldots, t_{m+n}^{(2)}\} \), referring to the text above. The parameters, along with global registers \( \{t_1^{(2)}, \ldots, t_{m+n}^{(2)}\} \) are used by the call and are not live afterwards; referring back to Fig. 7, they belong to \( C_1 \cap C_2 \); variables \( \{t_1^{(3)}, \ldots, t_{m+n}^{(3)}\} \), and any variables computed by the callee and returned to the caller are defined by the procedure, and belong to \( C_2 \setminus C_1 \). No variables are live across the call—recall: that is the entire point of launch and landing pads—so \( C_1 \cap C_2 \) is empty.

This proves the theorem by construction. □

As a corollary to Theorem 4, the spill test introduced by Pereira and Palsberg [13] can be applied to whole programs, where each procedure is converted to Elementary Form and extended launch and landing pads are inserted at call points.

VIII. EXPERIMENTAL RESULTS

A. Overview

This section provides an empirical comparison of our optimal algorithm for interprocedural register allocation—based on SSA Form with launch and landing pads—with the top-down and bottom-up color palette propagation (CPP) heuristics of Beidas and Zhu [16]. CPP is independent of the heuristic that colors the interference graph of each procedure.

Beidas and Zhu used Chaitin et al.’s [1, 2] heuristic to color each procedure’s interference graph. Chaitin et al.’s register allocator uses a greedy heuristic first published in 1876 by Kempe [32]. Matula and Beck [33] subsequently published an improvement to Kempe’s heuristic, called smallest last ordering (SLO), which became the basis for Briggs et al.’s [34] optimistic register allocator; Briggs et al.’s framework has subsequently been enhanced and improved by others [35, 36].

Our implementation of Beidas and Zhu’s framework uses SLO, rather than Kempe’s heuristic, and does not convert each procedure to SSA Form. We denote these top-down and bottom-up implementations as CPP-SLO\( \downarrow \) and CPP-SLO\( \uparrow \) respectively. We also evaluated CPP using SSA Form; each procedure is colored using Gavril’s optimal \( O(|V| + |E|) \)-time algorithm for chordal coloring [6], but without explicitly building an interference graph [7, 8]. We respectively denote our implementations as CPP-SSA\( \downarrow \) and CPP-SSA\( \uparrow \).

Lastly, we inserted launch and landing pads as discussed in Section VIII.A of this paper. This ensures that the IIG is a chordal graph and can be colored optimally; as a benefit, we have shown that each procedure’s interference graph can be colored individually, similar to CPP-SSA\( \uparrow \) and CPP-SSA\( \downarrow \). We denote this implementation as OPT, as it is optimal by Theorem 2.

CPP-SLO\( \downarrow \) and CPP-SLO\( \uparrow \) include a secondary pass that verifies the legality of the coloring, i.e., it checks for each edge \((u, v)\) that \( \text{color}(u) \neq \text{color}(v) \); in contrast to the SSA-based techniques, which do not build an interference graph, are correct-by-construction.

B. Experimental Setup and Benchmarks

We used the Machine SUFI compiler as our experimental platform. Although Machine SUFI is not a high-level synthesis tool, it provides us with all of the necessary features to study register allocation in an interprocedural context.

We selected a set of open-source embedded and multimedia applications written in C from the Mediabench [37] and MiBench [38] suites. The runtime of the Machine SUFI pass that links different files together and forms global symbol tables (link_suiF) was prohibitive in the general case, so the benchmarks studied here are among the smaller ones in the two suites. We did not implement pointer-analysis, so we only consider applications that do not contain function pointers.

The experiments were run on a Dell Latitude D810 laptop with an Intel Pentium M processor running at 2.0 GHz, with 1.0G of RAM; the operating system used was Fedora Core 3.

C. Results

Fig. 12(a) reports the number of registers allocated for each benchmark by the five interprocedural register allocation methods that we evaluated. Of the five methods, only OPT found the optimal solution—meaning the minimum number of registers—for every benchmark; CPP-SLO\( \uparrow \) found minimum register solutions for all benchmarks, except for gsm, mpeg2enc, and susan, and in each case, allocated only one more register than OPT; the other three heuristics found minimum register solutions much less often, and the disparity between OPT and the three other heuristic solutions was often greater than one register.

Fig. 12(b) reports the runtimes of the five techniques, normalized to the runtime of OPT. CPP-SLO\( \downarrow \) and CPP-SLO\( \uparrow \) run slower than the SSA-based techniques, because they must explicitly construct an interference graph, and verify the coloring afterwards.
The bottom-up CPP methods tended to run faster than the top-down methods. In the top-down approaches, colors are unavailable for every vertex in the interference graph for each procedure; in the bottom-up approaches, colors are only unavailable for variables that are live across specific call-points. The bottom-up approaches propagate information about unavailable colors to each vertex individually, whereas, the top-down approaches can propagate this information on the granularity of individual procedures.

IX. CONCLUSION

An optimal linear-time algorithm for interprocedural register allocation in high-level synthesis has been presented. The algorithm converts each procedure in the program to SSA Form, and then adds additional parallel copy operations—launch and landing pads—at each call point. This ensures that the IIG is a chordal graph, which can be colored optimally. By adopting the scalable color palette propagation methods of Beidas and Zhu [16], we can ensure optimality while coloring the interference graph for each procedure individually.

Extensions have been presented for two other SSA-based program representations—SSI Form and Elementary Form—which ensure that the IIG is respectively an interval or elementary graph. The extension for Elementary Form, in particular, shows that an optimal, polynomial-time spill test for use in register allocation in compilers [13] can be applied to whole programs without sacrificing optimality.

REFERENCES


His research interests include FPGAs, compilers, and design automation and architecture for application-specific processors. Dr. Brisky was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis (CASES), 2007, and the International Conference on Field Programmable Logic and Applications (FPL), 2009.

Dr. Brisky is or has been a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe, the IEEE Symposium on Application-Specific Processors, the International Workshop on Software and Compilers for Embedded Systems, and the Reconfigurable Architecture Workshop. He has been the general co-chair of the 4th IEEE Symposium on Industrial Embedded Systems, 2009, and will be the general co-chair of the 8th IEEE Symposium on Application Specific Processors, 2010.

Ajay K. Verma received the B.S. degree in computer science from the Indian Institute of Technology Kanpur, in 2003. Since 2004, he has been working toward the Ph.D. degree in the Processor Architecture Laboratory, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

His research interests include logic synthesis, optimization of arithmetic circuits, and design automation for application-specific processors. Mr. Verma was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis in 2007.

Paolo Ienne (S’94–M’96) received the Dottore degree in electronic engineering from the Politecnico di Milano, Milan, Italy, in 1991, and the Ph.D. degree from the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 1996. In December 1996, he joined the Semiconductors Group of Siemens AG, Munich, Germany (which later became Infineon Technologies AG). After working on datapath generation tools, he became the Head of the embedded memory unit in the Design Libraries division. In 2000, he joined EPFL, where he is currently a professor and heads the Processor Architecture Laboratory (LAP). His research interests include various aspects of computer and processor architecture, computer arithmetic, reconfigurable computing, and multiprocessor systems-on-chip.

Dr. Ienne is or has been a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe, the International Conference on Computer Aided design, the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, the International Symposium on Low Power Electronics and Design, the International Symposium on High-Performance Computer Architecture, the International Conference on Field Programmable Logic and Applications, and the IEEE International Symposium on Asynchronous Circuits and Systems. Ienne has been General Co-chair of the Sixth IEEE Symposium on Application-Specific Processors (SASP08) and a Guest Editor for a special section of the IEEE Transactions on Very Large Scale Integration (VLSI) Systems on Application Specific Processors. He was a recipient of the DAC 2003 and the CASES 2007 Best Paper Awards.

Philip Brisk received the B.S., M.S., and Ph.D. degrees, all in computer science, from UCLA in 2002, 2003, and 2006 respectively. From 2006-2009 he was a postdoctoral scholar in the Processor Architecture Laboratory in the School of Computer and Communication Sciences at the École Polytechnique Fédérale de Lausanne, (EPFL), in Lausanne, Switzerland. He is now an assistant professor in the Department of Computer Science and Engineering in the Bourns College of Engineering at the University of California, Riverside.