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

Philip Brisk, *Member, IEEE,* Ajay K. Verma, and 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 nondeterministic polynomial time-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*—Chordal graph, graph coloring, high level synthesis, (inteprocedural) register allocation, static single assignment (SSA) form.**

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 nondeterministic polynomial time (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 probably 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 probably belong to known subclasses of chordal graphs. To present, 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 the International Conference on Computer Aided Design 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 present, 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

Manuscript received August 25, 2009; revised December 8, 2009. Date of current version June 18, 2010. This paper was recommended by Associate Editor R. Camposano.

Philip Brisk is with the Department of Computer Science and Engineering, Bourns College of Engineering, University of California, Riverside, CA 92521 USA (e-mail: philip@cs.ucr.edu).

A. K. Verma and P. Ienne are with the Processor Architecture Laboratory, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne CH-1015, Switzerland (e-mail: ajaykumar.verma@epfl.ch; paolo.ienne@epfl.ch).

Digital Object Identifier 10.1109/TCAD.2010.2049060

0278-0070/$26.00 © 2010 IEEE

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 present, 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 probably 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). Kurdahi 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 variables 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. 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 CFG, a directed graph $G_{CFG} = (N, E_{CFG}, entry, exit)$: N is the set of basic blocks, and E_{CFG} is the set of control flow edges, i.e., an edge $(n_1, n_2) \in E_{CFG}$ indicates that control may transfer from n_1 to n_2; and *entry* and *exit* are distinguished basic blocks, both of which are empty; for each basic block $n \in N_{CFG}$, 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. 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 CPG. Let P be the set of procedures in the program, where $P_1 \in P$ 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_i calls another procedure P_j directly, or a function pointer calls a set of procedures Q_j.

The CPG is a directed hypergraph $G_{CPG} = (V_{CPG}, E_{CPG})$, where $V_{CPG} = P \bigcup C$; 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_{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_{CPG}, along with a hyper-edge (c_k, Q_j) which points to each procedure $P_l \in Q_j$. 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, $\{(c_k, P_l) | P_l \in Q_j\}$ 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 $v \leftarrow \cdots$ is called a *definition* of v, and an operation $\cdots \leftarrow v$ 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 is 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_1, v_2) for every pair of interfering variables.

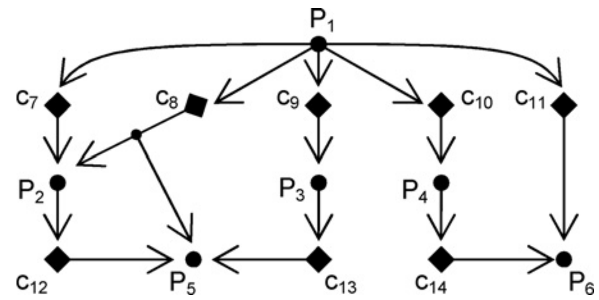

Fig. 1. Example call-points graph. This CPG contains one hyper-edge $(c_8, \{P_2, P_5\})$.

The interference graph, as defined above, is only applicable to an individual procedure. The 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 present.

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., $V = V_1 \cup V_2 \cup \ldots \cup V_{|P|}$, and E contains all global and local interferences, i.e., $E = E_1 \cup E_2 \cup \ldots, E_{|P|} \cup E_{global}$, where E_{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 of 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 $P_i^{(l)}, \ldots, P_i^{(d)}$ in the CPG; this would create d instances of $v, v^{(l)}, \ldots, v^{(d)}$, 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 Beidas and Zhu's [16] scalable heuristics for interprocedural register allocation; they introduced two heuristics, top-down, and bottom-up color palette propagation (CPP), 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_1 to P_i; we denote this quantity by δ_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 δ_i, it is straightforward to see that $\delta_j \geq \delta_i + |L(c_k)|$, as there is no guarantee that this path contains the maximum number of interprocedural interferences leading into P_j. 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 quantityδ_k with each call point c_k. Specifically, if c_k occurs in procedure P_i, then $\delta_k = \delta_i + |L(c_k)|$.

Inductively, the process works as follows. δ_1, corresponding to the entry procedure, is a constant. In principle, δ_1 may be zero, however, languages such as *C/C++* pass two parameters, *argc* and *argv* to *main*, so in this case, $\delta_1 = 2$. The specific constant used for δ_1 depends on the language, operating system, and or programmer decisions involved. This establishes the basis. For the induction hypothesis, assume that δ_i is known for each CPG vertex x such that the maximum distance from P_1 to x in the CPG is at most t hops. Now, consider a vertex $y \in V_{\text{CPG}}$ such that the maximum distance from P_1 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_1 to the caller P_i is at most t hops, so δ_i 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 $\delta_k = \delta_i + |L(c_k)|$.

(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_1, so δ_k is known. As every path from P_1 to P_j goes through precisely one call point in C_j, it follows that

$$\delta_j = \max_{c_k \in C_j} \{\delta_k\}. \quad (1)$$

Therefore, it follows that δ_j, corresponding to procedure P_j, is the maximum number of global interferences on any path from P_1 to P_j. Table I shows an example, corresponding to the CPG in Fig. 1; in this case, we assume that $\delta_1 = 0$, 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 δ-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_k. Recall that $L(c_k)$ is the set of variables that are live across c_k, and let *colors[L(c_k)]* denote

TABLE I
ALLOCATION OF GLOBAL REGISTERS FOR THE CPG IN FIG. 1 USING THE $|L(C_k)|$ VALUES PROVIDED IN THE SECOND COLUMN

Call Point	$\|L(C_k)\|$	δ_k	Procedure	δ_i
c_7	1	1	P_1	0
c_8	2	2	P_2	2
c_9	3	3	P_3	3
c_{10}	2	2	P_4	2
c_{11}	5	5	P_5	6
c_{12}	3	5	P_6	5
c_{13}	3	6		
c_{14}	2	4		

the set of colors that are assigned to the variables in $L(c_k)$. 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_l in P_j, the union of *colors[L(*c_k*)]* and *colors[L(*c_l*)]* are propagated to the procedure called from c_l.

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 χ_i denote the chromatic number of G_i. In this restricted case, it is straightforward to see that *exactly* δ_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

$$\chi(G) = \max_{P_i \in P} \{\delta_i + \chi_i\}. \tag{2}$$

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_4. For the sake of simplicity, assume that $L(c_3) = \{v_1\}$ and $L(c_4) = v_2$, and that $color(v_1) = 1$ and $color(v_2) = 2$. In this case, $\delta_2 = 1$, but two colors are unavailable on the palette, so the spectrum of colors used to color G_2 will be $\{3, 4, \ldots, \chi_2 + 2\}$. In principle, optimality could be guaranteed if, for every call point c_k that calls P_2, we can ensure that the palette of unavailable colors is $\{1, 2, \ldots, \delta_k\}$; 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 noninterfering vertices in the interference graph receive the same color. As coalescing is NP-hard for chordal graphs [19], top-down CPP 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 $\{1, 2, \ldots, \delta_j\}$. Details are provided in Section VI.

C. Bottom-Up Color Palette Propagation

Bottom-up CPP 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_k, then the colors assigned to variables in P_j, and every procedure reachable from P_j, are unavailable for variables in $L(c_k)$.

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 $\{1, 2, \ldots, \chi_2\}$ and $\{1, 2, \ldots, \chi_3\}$, respectively. To propagate these colors upward, we instantiate $\max\{\chi_2, \chi_3\}$ pre-colored vertices in G_1, the interference graph for P_1. Each vertex belonging to $L(c_4)$ and $L(c_5)$ interferes with the first χ_2 and χ_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 upward 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 φ-functions, which are placed at the beginning of basic blocks having multiple predecessors. In addition to φ-functions, SSI form requires a second type of parallel copy, called a π-function, which are placed at the end of basic blocks having multiple successors. In addition to φ- and π-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. φ-Functions

Fig. 2 illustrates φ-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 φ-function is placed at the entry point of the basic block following the two sides of the condition.

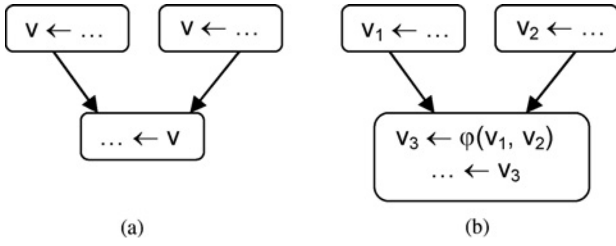

Fig. 2. (a) and (b) φ-Function is a copy operation inserted at the entry point of a basic block having multiple predecessors, during the conversion to SSA form. The φ-function merges two (or more) distinct definitions of a variable that converge at the entry point of the basic block. The copy operations are actually placed on the incoming edges of the block, rather than in the block itself; therefore, v_1 and v_2 do not interfere in (b).

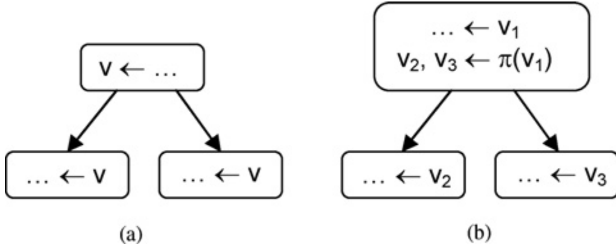

Fig. 3. (a) and (b) π-Function is a copy operation inserted at the exit point of a basic block having multiple successors, during the conversion to SSI form. The π-function splits a definition of a variable into two (or more) new variables that are used along different paths reachable from the basic block. The copy operations are actually placed on the incoming edges of the block, rather than in the block itself; therefore, v_2, and v_3 do not interfere in (b).

The φ-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 φ-functions instantiated for several variables at its entry point. A set of φ-functions can be represented as a matrix [8]

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_k \end{bmatrix} = \Phi \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & & x_{2m} \\ \dots & & & \dots \\ x_{k1} & x_{k2} & \dots & x_{km} \end{bmatrix}. \quad (3)$$

If control enters a basic block from its ith predecessor, then a parallel copy $[y_1, y_2, \dots, y_k] \leftarrow pcopy[x_{1i}, x_{2i}, \dots, x_{ki}]$ executes. Variables $y_1, y_2, \dots, y_k$ interfere with one another, as do $x_{1i}, x_{2i}, \dots, x_{ki}$; however, the parallel copy prevents any of the x_{ji}'s from interfering with the y_j's. Moreover, no x_{ji} variables in different columns interfere on account of the φ-function, but they may interfere if their lifetimes intersect elsewhere in the program.

B. π-Functions

Fig. 3 illustrates π-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 π-function is inserted to spit 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 φ-functions, a set of π-functions at the end of a basic block can be represented using matrix notation

$$\begin{bmatrix} y_{11} & y_{12} & \dots & y_{1m} \\ y_{21} & y_{22} & & y_{2m} \\ \dots & & & \dots \\ y_{k1} & y_{k2} & \dots & y_{km} \end{bmatrix} = \Pi \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_k \end{bmatrix}. \quad (4)$$

If control flows from n to its ith successor, then a parallel copy operation $[y_{1i}, y_{2i}, \dots, y_{ki}] \leftarrow pcopy[x_1, x_2, \dots, x_k]$ executed. Variables $x_1, x_2, \dots, x_k$ all interfere with one another, as do $y_{1i}, y_{2i}, \dots, y_{ki}$; however, the parallel copy prevents any of the y_{ji} from interfering with any of the x_j variables. Moreover, no two y_{ji} variables in different columns of the matrix interfere on account of the π-function, but they may interfere if their lifetimes intersect elsewhere in the program.

C. Parallel Copies Inside Basic Blocks

Both φ- and π-functions are parallel copy operations; φ-functions are effectively placed on the CFG edges that come into a basic block, while π-functions are effectively placed on outgoing edges. The elementary form [13] requires a third type of parallel copy operation in addition to φ- and π-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 φ- and π-functions are virtual entities. A compiler must eventually serialize them into nonparallel 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 φ- and π-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, $v_1, v_2, \dots, v_k$; then: 1) the union of the live ranges of $v_1, v_2, \dots, v_k$ is precisely the live range of v; 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

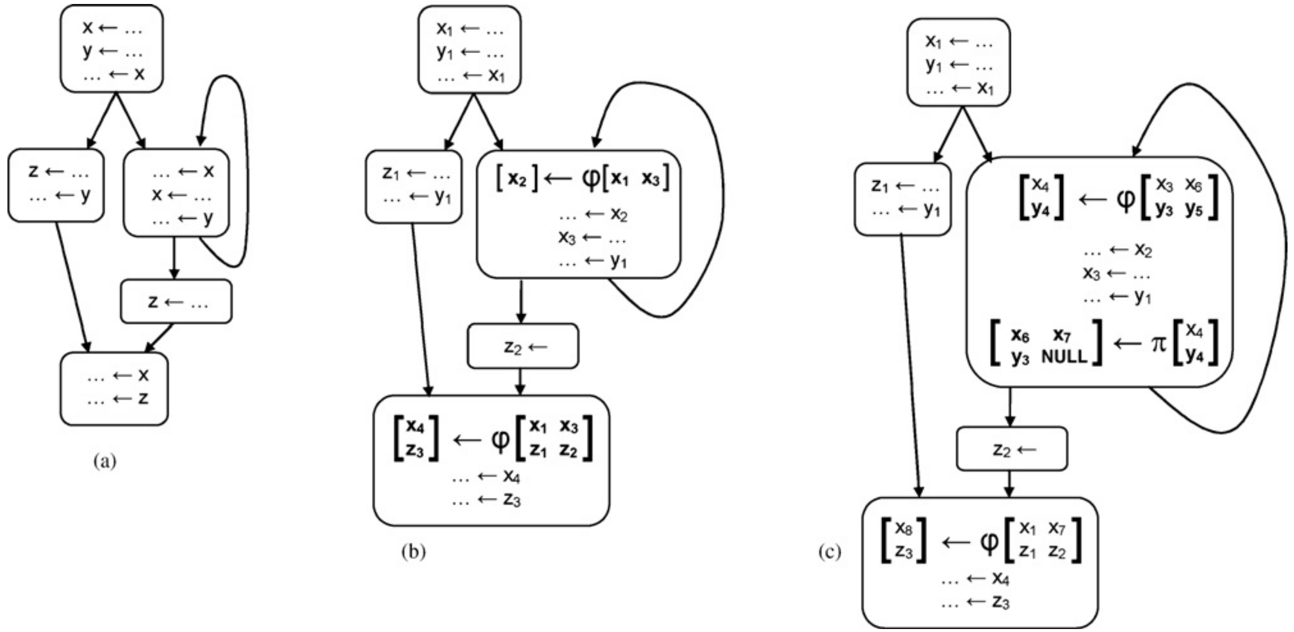

Fig. 4. Program fragment represented as (a) CFG, and converted to (b) SSA and (c) SSI form. The NULL definition in (c) occurs because variable y is not live outside of the loop in the CFG fragment in (a).

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 precisely 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 nonredundant parallel copies, and entails significantly more φ- and π-functions than SSA or SSI form. Specifically, a φ-function for variable v is placed at the entry point of *every* basic block having more than two predecessors where v is live; a π-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 $< v_0, v_1, \ldots, v_{k-1} >$ where $(v_i, v_{i+1 \bmod k}) \in E$, $0 \leq i \leq k-1$. A chord is an edge (v_i, v_j) between two nonadjacent vertices in the cycle, i.e., $j \neq i+1 \bmod k$ and $i \neq j+1 \bmod k$. 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 probably 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, $[low_v, high_v]$ 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 than 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].

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.

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. $C_1 \bigcap C_2$ is the set of variables that are live before and after the instruction. Without loss of generality, $C_1 \backslash C_2$ 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, $C_2 \backslash C_1$ is the set of variables that are not live

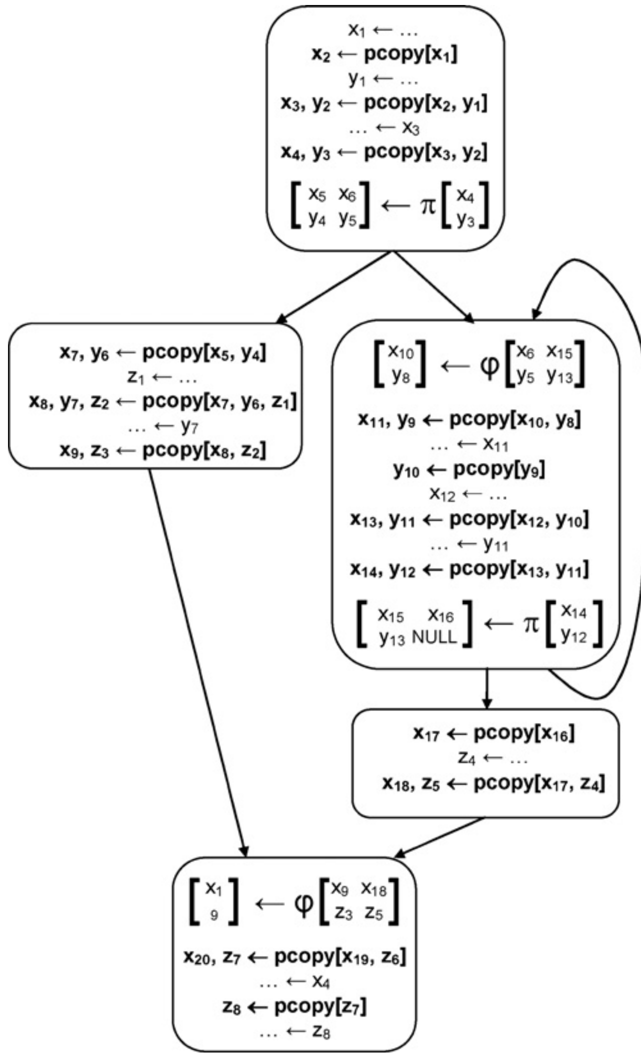

Fig. 5. Program fragment from Fig. 4 converted to elementary form.

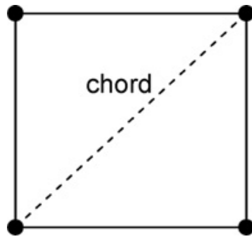

Fig. 6. 4-Cycle with a chord.

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 δ_i variables in any calling context leading from P_1 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 Section 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 δ_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 δ_i of these global registers. Alternatively, each variable defined locally in P_i would interfere with δ_i variables defined locally in other procedures for *every* unique path in the CPG leading to P_i.

Let $\delta_{\max} = \max\{\delta_i | 1 \leq i \leq |P|\}$; $\delta_{\max}$ is the maximum number of variables live across call points among every path in the CPG. Therefore, we allocate a set $T = \{t_1, t_2, \ldots, t_M\}$ of global registers, where $M = |T| = \delta_{\max}$. Now, consider a call point c_k where procedure P_i calls procedure P_j. To simplify notation, let $m = \delta_i$ and $n = |L(c_k)|$ 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 $\{t_1, \ldots, t_m\}$. P_i then copies the variables in $L(c_k)$ to the next subset of available global registers, $\{t_{m+1}, \ldots, t_{m+n}\}$, effectively *pushing* them onto a stack; in parallel P_i passes all of the necessary parameters to P_j. Next, P_j executes; upon completion, control returns to P_i, who *pops* the 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 Ψ and Ψ^{-1}, respectively.

Thus, a procedure call operates as follows:

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

$||$ *Pass parameters from* P_i *to* P_j

Call P_j

$(L'(c_k)) \leftarrow \Psi^{-1}(t_{m+1}, \ldots, t_{m+n})$.

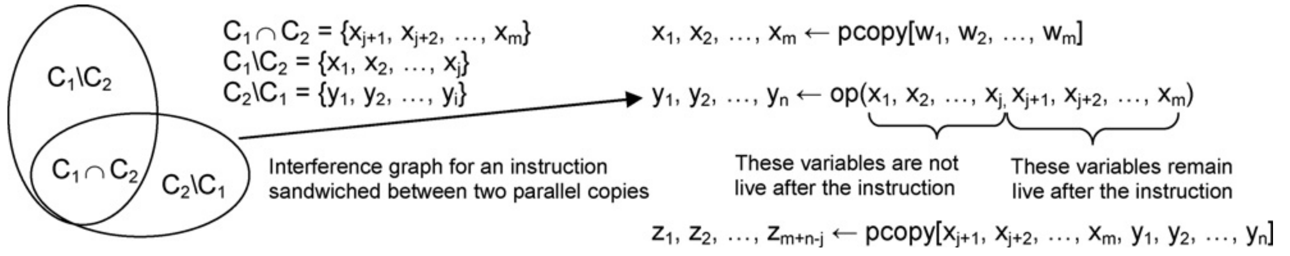

Fig. 7. In an elementary graph, each connected component is the interference graph for an individual instruction, which is the intersection of two cliques.

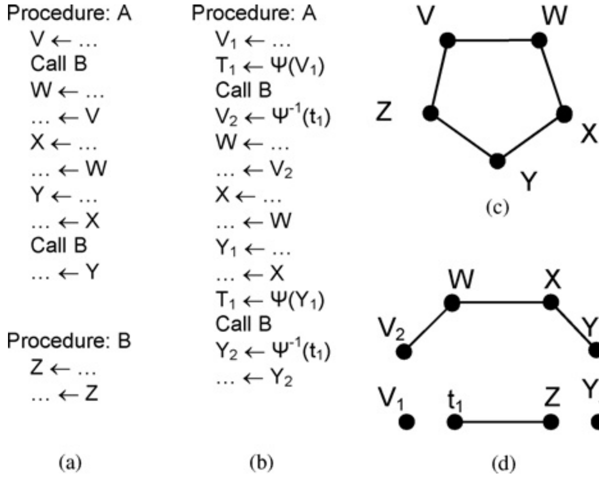

Fig. 8. (a) Small SSA form program. (b) After inserting launch and landing pads. (c) and (d) With respective interprocedural interference graphs.

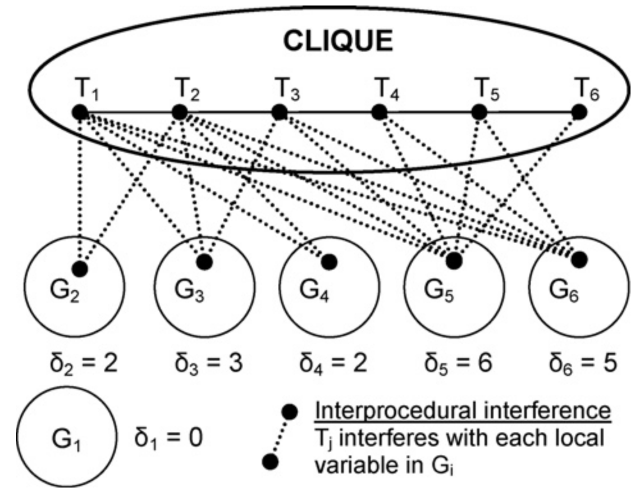

Fig. 9. Interprocedural interference graph for the program in Fig. 1 and Table I.

As the landing pad assigns new values to the variables in $L(c_k)$, they are renamed [denoted by the set $L'(c_k)$]; 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 $L(c_k)$ that is reachable from the landing pad is replaced with the corresponding renamed variable in $L'(c_k)$.

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*.

Fig. 8(c) and (d) shows the IIGs corresponding to the programs in Fig. 8(a) and (b). The IIG shown in Fig. 8(c) is a *5*-hole, which is nonchordal, 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_i can bind variables declared locally to any global register t_l, $l > \delta_i$. 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 two, 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 $t_k \in T$, so v_i and v_j cannot interfere: a contradiction. ■

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

Proof: Follows immediately from the fact that $\delta_{\max}$ is the maximum number of variables live across call points among every path in the CPG and $|T| = \delta_{\max}$. ■

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 $\{t_1, \ldots t_m\}$ where $m = \delta_i$. Let $\{G_i = (V_i, E_i) \mid 1 \leq i \leq N\}$ 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:

$$V = T \cup \bigcup_{i=1}^{N} V_i \tag{5}$$

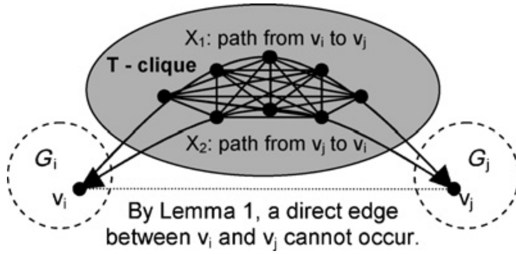

Fig. 10. Illustration of the proof of Theorem 2. If we assume that G is nonchordal, then a k-hole involving two global registers must be present. The k-hole involves paths X_1 from v_i to v_j and X_2 from v_j back to v_i, with no common vertices other than v_i and v_j. By Lemma 1, both of these paths must contain at least one global register; the edge between these two global registers is a chord.

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

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 $\{t_1, \ldots, t_m\}$, and no local variables defined in other procedures. Therefore, it suffices to color G_i using colors $\{m + 1, \ldots, m + \chi_i\}$, since $\{1, \ldots, m\}$ are unavailable. Equation (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 χ_i to the color assigned to each vertex. This approach is top-down, as the set of unavailable colors, $\{1, \ldots, \chi_i\}$, 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 nonchordal; 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 $t_j \in T$ is either adjacent to *every* vertex in V_i, or no vertices in V_i. Therefore, the k-hole must include vertices $v_i \in V_i$ and $v_j \in V_j$, corresponding to variables declared locally in distinct procedures. The k-hole is decomposed into paths $X_1 = (v_i, \ldots, v_j)$, and $X_2 = (v_j, \ldots, v_i)$, whose only common vertices are v_i and v_j. By Lemma 1, v_i and v_j 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.

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 δ-values, such that $\delta_i \leq \delta_{i+1}$ for every pair of consecutive intervals I_i, I_{i+1}. In accordance with Lemma 1, none of the intervals overlap. Let α_i be any value that satisfies $h_i < \alpha_i < l_{i+1}$ for $1 \leq i \leq N\text{-}1$, and let $\alpha_N = h_N + 1$. Global register t_k interferes with each variable defined locally in a procedure P_i having $\delta_i \geq k$. Therefore, we can find an index j such that $\delta_{j-1} < k$ and $\delta_j \geq k$; by convention, we let $\delta_0 = 0$. As the intervals have been sorted, t_k interferes with every variable defined locally in $\{P_j, \ldots, P_N\}$ and no variables defined locally in $\{P_1, \ldots, P_{j-1}\}$. Therefore, we associate the interval $I(t_k) = [\alpha_j, \alpha_N]$ with t_k. By construction, $I(t_k)$ subsumes the intervals $\{I_j, \ldots, I_N\}$ and does not overlap $\{I_1, \ldots, I_{j-1}\}$. 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 α_N. ■

Chordal graphs can be colored in $O(|V_i| + |E_i|)$ time [6]; interval graphs, in contrast, can be colored in $O(|V_i|)$ 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 present, 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_1 which interferes with every variable in V_i, 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

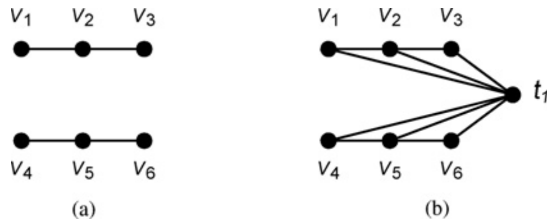

Fig. 11. (a) Interference graph G_i for an elementary form procedure P_i; G_i is an elementary graph. (b) If $\delta_i = 1$, and we build an IIG that includes global register t_1, the resulting graph is no longer an elementary graph. Specifically, this example indicates that no analogue of Theorems 2 and 3 exists for elementary graphs.

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 δ_i variables will be stored in global registers, depending on the calling context, and δ_i can be computed statically. When converting the callee to elementary form, each φ-function, π-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_k where P_i calls P_j, where $m = \delta_i$ and $n = |L(c_k)$. At the call point, assume that the names of the global registers are $\{t_l^{(1)}, \ldots, t_m^{(1)}\}$. 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 $L(c_k)$ to global registers $\{t_{m+1}, \ldots, t_{m+n}\}$, 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 $\{t_l^{(1)}, \ldots, t_m^{(1)}\}$ as well. We refer to this operation as an *extended launch pad*

$$(t_l^{(2)}, \ldots, t_m^{(2)}) \leftarrow pcopy[(t_1^{(1)}, \ldots, t_m^{(1)})]$$

$$||\, ({t_{m+1}}^{(2)}, \ldots, {t_{m+n}}^{(2)}) \leftarrow \Psi(L(c_k))$$

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

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

$$\textit{Call } P_j \,||\, (t_1^{(3)}, \ldots, {t_{m+n}}^{(3)}) \leftarrow pcopy[(t_1^{(2)}, \ldots, {t_{m+n}}^{(2)})].$$

In actuality, P_i is unaware of how many times P_j 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 $L'(c_k)$. 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*

$$(t_l^{(4)}, \ldots, t_m^{(4)}) \leftarrow pcopy[(t_l^{(3)}, \ldots, t_m^{(3)})]$$

$$||\, (L'(c_k)) \leftarrow \Psi^{-1}({t_{m+l}}^{(3)}, \ldots, {t_{m+n}}^{(3)}).$$

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 $t_j^{(k)}$ denote the *kth* copy of global register t_j, $1 \leq j \leq m$, that has been instantiated. By construction, $t_j^{(k)}$ 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 $C_1 \bigcap C_2$ is the set of variables that are live before and after the instruction; hence, $t_j^{(k)} \in C_1 \bigcap C_2$ 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_k; 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 $(t_1^{(3)}, \ldots, {t_{m+n}}^{(3)}) \leftarrow pcopy[(t_l^{(2)}, \ldots, {t_{m+n}}^{(2)})]$, referring to the text above. The parameters, along with global registers $(t_l^{(2)}, \ldots, {t_{m+n}}^{(2)})$ are used by the call, and are not live afterwards; referring back to Fig. 7, they belong to $C_1 \backslash C_2$; variables $(t_l^{(3)}, \ldots, {t_{m+n}}^{(3)})$, and any variables computed by the callee and returned to the caller are defined by the procedure, and belong to $C_2 \backslash C_1$. No variables are live across the call—recall: that is the entire point of launch and landing pads—so $C_1 \bigcap C_2$ 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 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

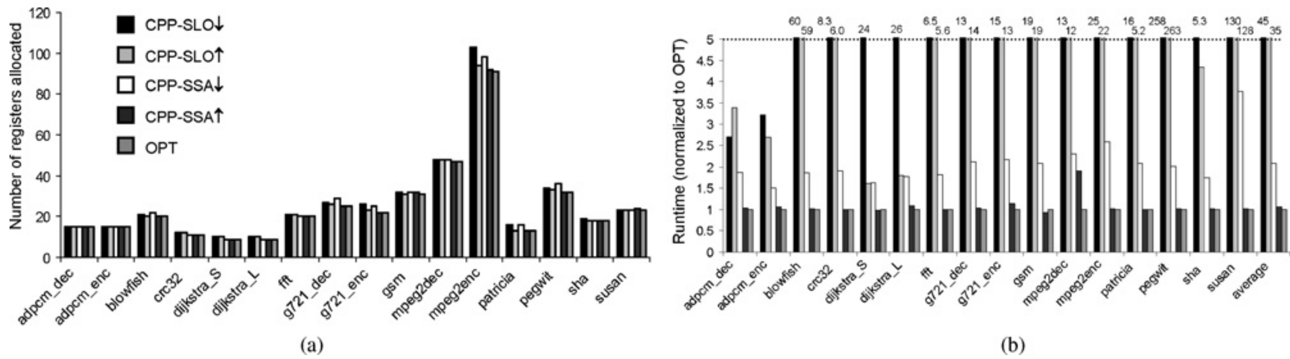

Fig. 12. Experimental results: (a) number of registers allocated by different interprocedural register allocation techniques; (b) runtime of the different interprocedural register allocation techniques (normalized to OPT).

[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↓ and CPP-SLO↑, 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↓ and CPP-SSA↑.

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↑ and CPP-SSA↓. We denote this implementation as OPT, as it is optimal by Theorem 2.

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

B. Experimental Setup and Benchmarks

We used the Machine SUIF compiler as our experimental platform. Although Machine SUIF 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 SUIF 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 1.0 G 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↑ 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↓ and CPP-SLO↑ 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 CPP 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

[1] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Comput. Langauges*, vol. 6, no. 1, pp. 47–57, 1981.

[2] G. J. Chaitin, "Register allocaiton and spilling via graph coloring," in *Proc. ACM SIGPLAN Symp. Compiler Construction*, 1982, pp. 98–101.

[3] C.-J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Comput.-Aided Design*, vol. 5, no. 3, pp. 379–395, Jul. 1986.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *Assoc. Comput. Mach. Trans. Program. Languages Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[5] D. L. Springer and D. E. Thomas, "Exploiting the special structure of conflict and compatibility grpahs in high level synthesis," *IEEE Trans. Comput.-Aided Design*, vol. 13, no. 7, pp. 843–856, Jul. 1994.

[6] F. Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph." *Soc. Ind. Appl. Math. J. Comput.*, vol. 1, no. 2, pp. 180–187, Jun. 1972.

[7] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh, "Optimal register sharing for high level synthesis of SSA form programs," *IEEE Trans. Comput.-Aided Design*, vol. 25, no. 5, pp. 772–779, May 2006.

[8] S. Hack and G. Goos, "Optimal register allocation for SSA-form programs in polynomial time," *Inform. Process. Lett.*, vol. 98, no. 4, pp. 150–155, May 2006.

[9] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation and spill complexity under SSA," ENS-Lyon, Lyon, France, Tech. Rep. 2005-33, 2005.

[10] C. S. Ananian, "The static single information form," M.S. thesis, Lab. Comput. Sci., Massachusetts Instit. Technol., Cambridge, Tech. Rep. MIT-LCS-TR-801, Sep. 1999.

[11] J. Singer, "Static program analysis based on virtual register renaming," Ph.D. dissertation, Computer Laboratory, Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-600, Feb. 2006.

[12] B. Boissinot, P. Brisk, A. Darte, and F. Rastello, "SSI revisited," ENS-Lyon, Lyon, France, Tech. Rep. LIP 2009-24, 2009.

[13] F. M. Q. Pereira and J. Palsberg, "Register allocation by puzzle solving," in *Proc. Int. Conf. Program. Language Design Implementation*, 2008, pp. 216–226.

[14] P. Brisk, A. K. Verma, and P. Ienne, "Optimal polynomial-time interprocedural register allocation for high level synthesis and ASIP design," in *Proc. Int. Conf. Comput.-Aided Design*, 2007, pp. 172–179.

[15] R. Vemuri, S. Katkoori, M. Kaul, and J. Roy, "An efficient register optimization algorithm for high level synthesis from hierarchical behavioral specifications," *Assoc. Comput. Mach. Trans. Design Autom. Electron. Syst.*, vol. 7, no. 1, pp. 189–216, Jan. 2002.

[16] R. Beidas and J. Zhu, "Scalable interprocedural register allocation for high level synthesis," in *Proc. Asia South Pacific Design Autom. Conf.*, 2005, pp. 511–516.

[17] F. J. Kurdahi and A. C. Parker, "REAL: A program for register allocation," in *Proc. Design Autom. Conf.*, 1987, pp. 210–215.

[18] L. Stok, "Transfer free register allocation in cyclic data flow graphs," in *Proc. Eur. Conf. Design Autom.*, 1992, pp. 181–185.

[19] M. Farach-Colton and V. Liberatore, "On local register allocation," *J. Algorithms*, vol. 37, no. 1, pp. 37–65, Oct. 2000.

[20] F. Bouchez, A. Darte, and F. Rastello, "On the complexity of spill everywhere under SSA form," in *Proc. ACM SIGPLAN/SIGBED Conf. Languages Compilers Tools Embedded Syst.*, 2007, pp. 103–112.

[21] F. Bouchez, A. Darte, and F. Rastello, "On the complexity of register coalescing" in *Proc. Int. Symp. Code Generation Optimization*, 2007, pp. 102–114.

[22] M. Biró, M. Hujter, and Z. Tuza, "Precoloring extension I: Interval graphs," *Discrete Math.*, vol. 100, nos. 1–3, pp. 267–279, May 1992.

[23] D. Marx, "Precoloring extension on unit interval graphs," *Discrete Appl. Math.*, vol. 154, no. 6, pp. 995–1002, Apr. 2006.

[24] J. Runeson and S.-O. Nyström, "Retargetable graph-coloring register allocation for irregular architecture," in *Proc. Int. Workshop Software Compilers Embedded Syst.*, 2003, pp. 240–254.

[25] M. D. Smith, N. Ramsey, and G. H. Holloway, "A generalized algorithm for graph-coloring register allocation," in *Proc. Int. Conf. Program. Language Design Implement.*, 2004, pp. 277–288.

[26] F. Chow, "Minimizing register usage penalty at procedure calls," in *Proc. Int. Conf. Program. Language Design Implement.*, 1988, pp. 85–94.

[27] S. M. Kurlander and C. N. Fischer, "Minimum cost interprocedural register allocation," in *Proc. ACM SIGPLAN/SIGACT Symp. Principles Program. Languages*, 1996, pp. 230–241.

[28] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren, "Conversion of control dependence to data dependence," in *Proc. ACM Symp. Principles Program. Languages*, 1983, pp. 177–189.

[29] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August, "A framework for unrestricted whole program optimization," in *Proc. ACM SIGPLAN Conf. Program. Language Design Implement.*, 2006, pp. 61–71.

[30] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *Proc. ACM SIGPLAN/SIGACT Symp. Principles Program. Languages*, 1991, pp. 55–66.

[31] S. Zhang and W. M. Dai, "Linear-time left edge algorithm," in *Proc. Int. Conf. Chip Design Autom.*, 2000.

[32] A. B. Kempe, "On the geographical problem of the four colors," *Am. J. Math.*, vol. 2, pp. 193–200, 1879.

[33] D. W. Matula and L. L. Beck, "Smallest last-ordering and clustering graph coloring algorithms," *J. Assoc. Comput. Mach.*, vol. 30, no. 3, pp. 417–427, Jul. 1983.

[34] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *Assoc. Comput. Mach. Trans. Program. Languages Syst.*, vol. 16, no. 3, pp. 428–455, May 1994.

[35] L. George and A. W. Appel, "Iterated register coalescing," *Assoc. Comput. Mach. Trans. Program. Languages Syst.*, vol. 18, no. 3, pp. 300–324, May 1996.

[36] J. Park and S.-M. Moon, "Optimistic register coalescing," *Assoc. Comput. Mach. Trans. Program. Languages Syst.*, vol. 26, no. 4, pp. 735–765, Jul. 2004.

[37] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 330–335.

[38] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Workshop Workload Characterization*, 2001, pp. 3–14.

Philip Brisk (M'09) received the B.S., M.S., and Ph.D. degrees, all in computer science, from the University of California, Los Angeles, in 2002, 2003, and 2006, respectively.

From 2006 to 2009, he was a Post-Doctoral Scholar with the Processor Architecture Laboratory, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Bourns College of Engineering, University of California, Riverside. His current research interests include field-programmable gate arrays, compilers, and design automation and architecture for application-specific processors.

Dr. Brisk was a recipient of the Best Paper Award at both the 2007 International Conference on Compilers, Architecture, and Synthesis, and 2009 International Conference on Field Programmable Logic and Applications. He is or has been a member of the program committees of several international conferences and workshops, including the Design Automation and Test in Europe, IEEE Symposium on Application-Specific Processors, International Workshop on Software and Compilers for Embedded Systems, and Reconfigurable Architecture Workshop. He was the General co-Chair of the 4th IEEE Symposium on Industrial Embedded Systems in 2009, and will be the General co-Chair of the 8th IEEE Symposium on Application Specific Processors in 2010.

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

His current 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 2007 International Conference on Compilers, Architecture, and Synthesis.

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 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. Since 2000, he has been with the EPFL, where he is currently a Professor and heads the Processor Architecture Laboratory. His current research interests include various aspects of computer and processor architecture, computer arithmetic, reconfigurable computing, and multiprocessor systems-on-chip.

Dr. Ienne was a recipient of the Best Paper Awards at DAC 2003 and CASES 2007. He is or has been a member of the program committees of several international conferences and workshops, including the Design Automation and Test in Europe, International Conference on Computer-Aided Design, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, International Symposium on Low-Power Electronics and Design, International Symposium on High-Performance Computer Architecture, International Conference on Field Programmable Logic and Applications, and IEEE International Symposium on Asynchronous Circuits and Systems. He was the General co-Chair of the 6th IEEE Symposium on Application-Specific Processors in 2008, and a Guest Editor for a special section of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS ON APPLICATION-SPECIFIC PROCESSORS.