Optimal Polynomial-Time Interprocedural Register Allocation for High-Level Synthesis and ASIP Design

Philip Brisk Ajay K. Verma Paolo Ienne
Processor Architecture Laboratory
Ecole Polytechnique Federale de Lausanne
Lausanne, Switzerland
{philip.brisk, ajaykumar.verma, paolo.ienne}@epfl.ch

***Abstract*—Register allocation, in high-level synthesis and ASIP design, is the process of determining the number of registers to include in the resulting circuit or processor. The goal is to allocate the minimum number of registers such that no scalar variable is spilled to memory. Previously, an optimal polynomial-time algorithm for this problem has been presented for individual procedures represented in *Static Single Assignment (SSA) Form*. This result is now extended to complete programs (or sub-programs), as long as: (1) each procedure is represented in SSA Form; and (2) at every procedure call, all live variables are split at the call point. With this representation, it is possible to ensure that the interprocedural interference graph (IIG) is chordal, and can therefore be colored optimally in polynomial time. An optimal coloring of the IIG can be achieved by allocating registers for each procedure individually. Previous work has shown that optimal register allocation in SSA Form does not require an interference graph. Optimal interprocedural register allocation, therefore, is achieved without constructing an interference graph, giving the optimal algorithm a significant runtime advantage over prior sub-optimal heuristics.**

I. Introduction

Time-to-market is incredibly important for embedded system designers. To reduce time-to-market, automated techniques have been developed to generate customized hardware from a high-level description of an application. The search space for an application is infinitely large, which makes tool development in this area a daunting task. The most general version of this problem is *high-level synthesis*, where a user provides: (1) an application; and (2) an objective function and set of constraints, typically related to performance or power consumption. Given this information, the synthesis tool then attempts to construct an ideal hardware design. Due to the large search space, user-intervention is often required; fortunately, many tasks during synthesis have successfully been automated in the past.

Because of the complexity of high-level synthesis, many embedded systems are realized as *Application-Specific Instruction set Processors (ASIPs)*. An ASIP resembles a typical processor, but is tailored specifically to the application: unused operations are not supported; hardware accelerators are typically included; and architectural parameters such as register file size and scratchpad or cache memory size, are designed to meet the needs of an application. Although the ASIP design space is quite large, it is nonetheless much more limited than high-level synthesis, and automation techniques in this area—which are typically derived from compiler optimizations—are much better understood. This reduces the burden placed on the system designer and can significantly reduce time-to-market.

One important problem in both high-level synthesis and automated ASIP design is *register allocation*: the process of determining how many registers should be included in an embedded system [1, 7 19, 30-31]. A similar problem exists in compilers, where the number of registers in the target architecture is fixed [5, 8]. The compiler analogue of this problem is NP-Complete [3-4, 12]. In synthesis, in contrast, the goal is to allocate the smallest possible number of registers that can store all scalar variables, while non-scalar variables—linked lists and arrays—are stored in memory. This paper focuses on the problem of interprocedural register allocation in the context of high-level synthesis and automated ASIP design. The contribution, limitations, and related assumptions are stated succinctly in the next three subsections.

A. *Contribution*

We prove that the interprocedural register allocation for high-level synthesis and ASIP design has an optimal, polynomial-time solution. The proof extends prior work on interprocedural register allocation by Vemuri et al. [31] and Beidas and Zhu [1] that solved the problem sub-optimally using heuristics. To achieve an optimal solution, each procedure must be represented in SSA Form [5]; furthermore, all of the variables that are live across each procedure call must be split across the call point; we call the resulting representation *SSA with Launch and Landing Pads (SSA-LLP) Form*.

The implementation of the proposed register allocation algorithm is quite straightforward and can be implemented in three simple steps. The first is to determine how many registers are necessary to store variables across procedure calls, and is described in Section III.B; this step is implemented using a topological sort of a DAG. The second step, in Section IV, is to insert launch and landing pads before and after each procedure call. Third, the chordal interference graph for each SSA Form procedure, is colored, but with an offset, as described in the proof of Theorem 3 in Section V.D. The algorithm to color a chordal graph $G = (V, E)$ is a simple greedy algorithm [13].

B. *Limitations*

The claim of optimality in this paper is limited, based on the following assumptions:

(1) We assume that *static local variables* reside in memory. These variables are assigned a value on the first call to a procedure, and retain their values across multiple calls. Interprocedural liveness analysis for static variables is left open for future work.

(2) Pointer resolution is NP-Complete [23, 32], so interprocedural register allocation becomes NP-Complete if there are function pointers. The algorithm presented here can handle function pointers, as long as points-to sets are resolved in advance.

C. Assumptions Regarding Recursion

Non-static local variables must be pushed onto a stack across recursive procedure calls; otherwise, value of the variable would be overwritten when the procedure is re-entered during the next recursive call. Taken in conjunction with Limitation (1) above, we can safely assume that no variables that are live across recursive function calls reside in registers.

Fixed-depth recursion is not discussed in this paper; however, a stack of bounded size could be implemented with registers, if desired.

II. Related Work

Here, we summarize work on register allocation in high-level synthesis and the design of ASIPs.

A. Register Allocation Overview

In compilers and high-level synthesis, register allocation is modeled as a graph coloring problem. An *interference graph* $G = (V, E)$ is constructed, where each vertex $v \in V$ represents a variable, and an edge (x, y) is added to E if the lifetimes of variables x and y overlap. Two *interfering* variables cannot share the same storage location. An *Interprocedural Interference Graph (IIG)* [31] is an undirected graph $G = (V, E)$, where there is a vertex in V for every variable in the program, and an edge (x, y) is placed between each pair of variables x and y that interfere locally or globally (i.e. across procedure calls). The interference graph for each procedure is a subgraph of the IIG by construction.

In high-level synthesis and ASIP design, register allocation is solved by computing the chromatic number of the interference graph, $\chi(G)$. All variables corresponding to vertices that receive the same color are assigned to the same register. Although graph coloring is NP-Complete, polynomial-time solutions exist for certain classes of graphs, including *chordal graphs* (the subject of this paper), *interval graphs*, and *comparability graphs*, which are mentioned in the context of related work.

B. Register Allocation in High-Level Synthesis

Tseng and Siewiorek [30] were the first to model register allocation in high-level synthesis as a graph problem akin to coloring. Kurdahi and Parker [19] showed that interference graphs for scheduled data flow graphs are interval graphs. Springer and Thomas [27] showed that chordal interference graphs arise when restrictions are placed on variable lifetimes and procedure calls.

Independently, Brisk et al. [7], Bouchez et al. [2], and Hack and Goos [16] proved that an interference graph for a procedure in SSA Form is chordal. Since the conversion to SSA Form does not increase the chromatic number of an interference graph, this ensures an optimal polynomial-time solution for register allocation.

Interprocedural register allocation in high-level synthesis was studied by Vemuri et al. [30] and Beidas and Zhu [1]. Vemuri et al. constructed an IIG, which includes local interferences (variables within the same procedure) and global interferences across procedure calls, and colored it with a heuristic. Beidas and Zhu were concerned with the runtime and memory cost associated with the construction of the IIG. They developed a scalable algorithm, *Color Palette Propagation (CPP)*, which avoids the construction of an IIG and colors each procedure individually. Top-down and bottom-up CPP heuristics propagate interferences across procedure call boundaries. Their results were comparable to Vemuri et al., with a runtime that was approximately 100× faster. The optimal algorithm presented here is special case of top-down CPP; however, the program is represented in SSA-LLP Form, and a pre-processing stage is necessary to pre-allocate *global registers* that hold variables that are live across procedure calls.

C. Save-Free Interprocedural Register Allocation

There is a similarity between this paper and the work by Kurlander and Fischer [20] on *save-free interprocedural register allocation* in compilers. First, they perform register allocation—spilling and register assignment—locally for each procedure, which ensures that the interference graph for each procedure is a clique. Next, they construct an IIG over the call graph, which has the structure of a comparability graph. A maximum weight *k*-antichain is computed, which determines which sets of variables reside in registers or memory. Because interferences across procedure calls are not considered during their register assignment phase, it seems highly unlikely that their approach could be coaxed into providing an optimal solution for the problem addressed in this paper.

D. Register Assignment

Somewhat beyond the scope of this paper is the problem of register assignment. There may be many different minimum colorings of an interference graph. In compilers, as well as ASIP design, the goal of register assignment is to eliminate as many copy instructions, $y \leftarrow x$ as possible by assigning x and y to the same register. This problem was proven NP-Complete by Bouchez et al. [3]; an optimal solution, based on integer linear programming, was described by Grund and Hack [14]. In high-level synthesis, the goal of register assignment (a special case of *connectivity binding*) is to minimize the cost of interconnect—wires and multiplexers; Pangrle [24] proved this problem to be NP-Complete. It should be noted that due to interconnect cost, increasing the number of registers can lead to reduced chip area. Consequently, in the context of synthesis, an optimal allocation of registers using the technique described in this paper is only a starting point, and may not be the best solution.

E. Compiler Preliminaries

Compiler concepts such as *liveness analysis* and interference graph construction can be found in a compiler textbook (e.g. [11]). A good overview of SSA Form is presented by Briggs et al. [5].

III. Pre-Allocation of Global Registers

The first step is to compute the number of registers, M, necessary to hold variables that are live across call points, on every execution path in the program. We then allocate a set of *global registers*, $T = \{T_1, \ldots, T_M\}$, to hold these values. Once these registers have been allocated, the program can be converted to SSA-LLP Form, as discussed in Section IV.

A. The Call Points Graph (CPG)

Let P be the procedures in an program, and C be the set of points in the program where one procedure calls another. The *Call-Points Graph (CPG)* is a directed graph, $G_{CPG} = (V_{CPG}, E_{CPG})$, where $V_{CPG} = P \cup C$. Let $c_k \in C$ be a call point where procedure P_i calls procedure P_j. Then edges (P_i, c_k) and (c_k, P_j) are added to E_{CPG}. If there are no function pointers, then each call point has exact one predecessor (the caller) and one successor (the callee) in the CPG; in the case of function pointers, a call point will have multiple successors: each procedure in its points-to set. We assume that P_1 is the entry procedure of the program (typically called *main* in languages like *C/C*++), and that P_1 is the only node in the CPG with no predecessors. An example CPG is shown in Fig. 1.

A cycle in the CPG represents a set of mutually recursive functions. For our purpose, it suffices to eliminate recursive function calls from the CPG since locally defined variables will reside in memory across these call points. We compute the *strongly connected components (SCCs)* [28] of the CPG, and collapse each SCC into a single node. The result is called the *Augmented Components Graph*. We assume that the CPG is acyclic to simplify the discussion.

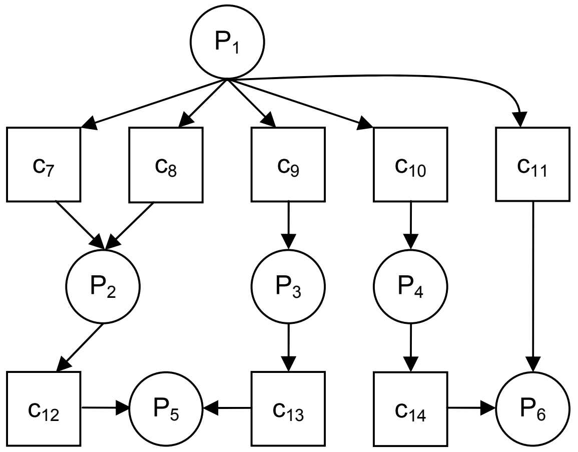


Figure 1. An example CPG.

B. *Allocation of Global Registers*

In interprocedural register allocation, we must determine how many registers are necessary to store variables that are live across each call on each path through the CPG. Let P_i be a procedure with interference graph $G_i = (V_i, E_i)$, and let the chromatic number of G_i be $\chi_i = \chi(G_i)$. If δ_i is the number of global interferences between P_i and local variables defined within its ancestors in the CPG, then $\delta_i + \chi_i$ registers are needed for P_i. If P_i is represented in SSA Form, then G_i is chordal and χ_i is computed efficiently. Here, we describe how to compute δ_i efficiently as well.

Let c_k be a call point in the CPG where P_i calls P_j. Let $L(c_k)$ be the set of variables in P_i that are live across c_k. Let $T = (P_1, P_2, \ldots, P_j)$ be a path in the CPG from P_1 to P_j. To simplify notation, let $C = \{c_1, \ldots, c_{k-1}\}$ be the call points along this path. Then the set of all variables live along path T when P_j is called is

$$L(T) = \bigcup_{i=1}^{k-1} L(c_i). \qquad (1)$$

The number of registers required to store variables along this particular path is $|L(T)|$. One way to compute δ_j would be to enumerate every path from P_1 to P_j and select the largest *L-value* among these paths; however, there are an exponential number of unique paths in a DAG in the worst-case. Instead, we can compute δ_j in $O(|V_{CPG}| + |E_{CPG}|)$ time by processing the basic blocks of the CPG in topological order. First, let us redefine δ as a function $\delta: V_{CPG} \rightarrow \{0, 1, \ldots\}$; we use the notation δ_j in place of $\delta(P_j)$ for brevity. For a procedure P_j, δ_j is defined as described above; $\delta_1 = 0$, since there are no variables live across the entry procedure. For a call point c_k where P_i calls P_j, $\delta_k = \delta_i + |L(c_k)|$.

For procedure P_i, let C_i be the set of call points that call P_i. Since vertices are processed in topological order, δ_i is known before we compute δ_k for each call point $c_k \in C_i$. Then

$$\delta_i = \max_{c_k \in C_i} \{\delta_k\}. \qquad (2)$$

The correctness of this algorithm follows from the fact that the CPG is acyclic and the vertices are processed in topological order; a formal proof is omitted to save space. An example, corresponding to Fig. 1, is shown in Table I. The $|L(c_i)|$ values are chosen arbitrarily in this example; one cannot determine, from a call graph alone, how many variables are live across each procedure call.

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

Call Point	$\|L(c_i)\|$	δ_i	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		

Let $\delta_{max} = max\{\delta_1, \ldots, \delta_N\}$, where $N = |V_{AC\text{-}CPG}|$. A *leaf* is a procedure with no successors in the CPG, for example, P_5 and P_6 in Fig. 1. It is easy to see that δ_{max} corresponds to an δ_i-value for at least one leaf. One δ_{max} is known, we allocated a set of global registers $T = \{T_1, \ldots, T_M\}$ to hold variables that are live across procedure calls. Parallel copy operations, described in Section IV, are inserted before and after each procedure call to ensure that the variables that are live across the call are assigned to a register in T.

C. *Handling Function Pointers*

Here, we discuss how to allocate global registers in the presence of function pointers. Since function pointer resolution is NP-Complete [23, 32], interprocedural register allocation is NP-Complete in their presence; however, if all function pointers are resolved in advance, then the problem can be solved in polynomial time. For each indirect call point c_k, let $PointsTo(c_k)$ be the set of functions that can be called from c_k; $PointsTo(c_k)$ has already been resolved in advance. The CPG can be constructed as described previously, but with the following change: c_k no longer has a single successor; instead, each procedure $P_j \in PointsTo(c_k)$ is a successor of c_k. Other than that, the computation of δ_i remains the same.

IV. SSA-LLP FORM

The set of global registers, T, allocated in section III.B, is used to hold variables that are live across procedure calls. To make this explicit, we explicitly copy each variable that is live across a procedure call to a global register, and then copy the value back to the variable immediately after the call. The justification and process for doing this is described next.

Consider procedure P_i. We assume that prior to calling P_i then $m = \delta_i$ variables that are live at the point where P_i is called reside in registers $T_1, \ldots T_m$. Now, consider a call point c_k where P_i calls P_j. At the call point, we have an additional $n = \delta_k - \delta_i$ variables that are live across the call. These variables are stored in registers $T_{m+1}, \ldots, T_{m+n}$.

We cannot assume that the color assignment phase will be able to assign these variables to the desired registers. Instead, we introduce parallel copy instructions—*Launch* and *Landing Pads*, respectively—before and after each call instruction respectively. Launch pads copy the variables in $L(c_k)$ to global registers $T_{m+1}, \ldots, T_{m+n}$, and landing pads copy them back to their original registers. Ψ denotes a launch pad and Ψ^{-1} denotes a landing pad. A procedure call augmented with launch and landing pads would have the following form:

$$\begin{aligned} &(T_{m+1}, \ldots, T_{m+n}) \leftarrow \Psi(L(c_k)) \\ &Call\ P_j \qquad (3) \\ &(L(c_k)) \leftarrow \Psi^{-1}(T_{m+1}, \ldots, T_{m+n}) \end{aligned}$$

Launch and landing pads eliminate all interferences between variables defined locally in separate procedures. Global interferences are now between a variable assigned to a register in T and a variable defined locally in another procedure further down the call chain.

Ψ and Ψ^{-1}, as stated previously, are parallel copy operations, similar to φ-functions in SSA Form [5]. In high-level synthesis, these copies become direct register-to-register transfers; in ASIP design, the copies can be serialized after register assignment [5, 16]. In both cases, effective register assignment, which is beyond the scope of this paper, can eliminate (some of) the overhead that arises due to these copies.

There is a similarity between the launch and landing pads and caller-save registers [9] used for interprocedural register allocation in compilers. Specifically, all registers, except those in T, are caller-save, in this context, and the registers in T receive the values immediate prior to the call. In a typical compiler, the variables would be pushed and popped onto the stack frame rather than copied to and from registers in T.

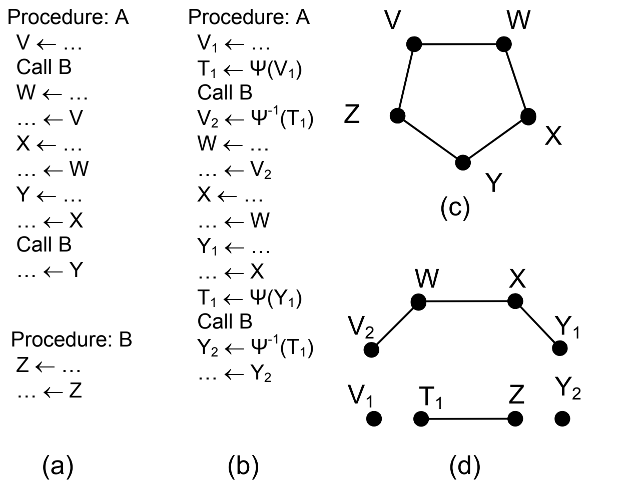
A. Preserving the Static Single Assignment Property

Any instruction of the form $y \leftarrow \ldots$ *defines* variable y. One property of SSA Form is that variables are defined once [5]. The conversion to *SSA Form with Launch and Landing Pads (SSA-LLP)*, however, violates this constraint. Each variable in $L(c_k)$ is defined multiple times: once at the original definition point, and now by a landing pad. This can easily be rectified by inserting launch and landing pads prior to SSA construction, or running the renaming phase of SSA construction [5] for variables that now have multiple definitions due to the insertion of landing pads.

Since register allocation is the only optimization that requires launch and landing pads, there is no need to insert launch and landing pads in advance, so all other optimizations and analyses can proceed uninhibited. Alternatively, the compiler can treat the launch pad, call instruction, and landing pad as one atomic operation that is not exposed to the optimizer; this hides both the re-definition of variables and the set of global registers until register allocation.

B. Example

Fig. 2 shows an example that illustrates the use of launch and landing pads, and shows that the conversion to SSA-LLP Form can reduce the chromatic number of an IIG.



```
Procedure: A
  V ← …
  Call B
  W ← …
  … ← V
  X ← …
  … ← W
  Y ← …
  … ← X
  Call B
  … ← Y

Procedure: B
  Z ← …
  … ← Z
```

(a)

```
Procedure: A
  V1 ← …
  T1 ← Ψ(V1)
  Call B
  V2 ← Ψ-1(T1)
  W ← …
  … ← V2
  X ← …
  … ← W
  Y1 ← …
  … ← X
  T1 ← Ψ(Y1)
  Call B
  Y2 ← Ψ-1(T1)
  … ← Y2
```

(b)

Figure 2. A small SSA Form program (a); converted to SSA-LLP Form (b), with respective IIGs (c) and (d).

Fig. 2 (a) shows a short program containing two functions, A and B, both of which satisfy the criteria for SSA Form. Fig. 2 (b) shows procedure A converted to SSA-LLP Form (with renaming, to preserve SSA); procedure B calls no functions, so it needs no launch or landing pads. The respective IIGs are shown in Fig. 2 (c) and (d) respectively. The IIG in Fig. 2 (c) is a well-known graph called a *5-hole* [10]. The 5-hole is well-known because it is the smallest *imperfect* graph, i.e., one whose chromatic number is larger than the cardinality of its maximal clique. The largest clique contains 2 vertices, but its chromatic number is 3. After converting procedure A to SSA-LLP form, the resulting IIG is shown in Fig. 2 (d). It's chromatic number is 2 rather than 3; thus converting to SSA-LLP form reduced the number of allocated registers. Moreover, the IIG in Fig. 2 (d) is a chordal graph. Chordal graphs are introduced in the next section.

V. Characterizing and Coloring the IIG

Here, we prove that the IIG for an application in SSA-LLP Form is a chordal graph and describe how to color it.

A. Chordal Graphs

Let $G = (V, E)$ be an undirected graph. A *cycle* is a set of vertices $\{v_0, v_1, \ldots, v_j\}$ in G, such that for $i = 0, 1, \ldots, j$, there is an edge $(v_i, v_{(i+1) \bmod j})$. A *chord* is an edge (v_s, v_t) that is not part of the cycle, i.e., if $s < t$, then $t \neq (s+1) \bmod j$. A *chordal graph* is a graph that contains no *chordless cycles (holes)* of length 4 or more. The IIG in Fig. 2 (d), for example, is chordal, while the IIG in Fig. 2 (c) is not.

For $v \in V$, $N(v)$ is the set of vertices adjacent to v. An *Elimination Order (EO)* is a function σ that assigns a unique number from the set $\{1, .., |V|\}$ to each vertex. Given an EO, vertices are named such that $\sigma(v_i) = i$. Let G_0 be the empty graph, $V_i = \{v_j \mid j \leq i\}$, and $G_i = (V_i, E_i)$ be the subgraph of G induced by V_i. $N_i(v_j) = \{v_k \in N(v_j) \mid k < i\}$. In other words, $N_i(v_j)$ contains all vertices adjacent to v_j that occur before v_j in the EO. Starting with G_0, an EO allows the complete graph G to be reconstructed one vertex at a time.

Vertex v is *simplical* if $N(v)$ is a clique. A *Perfect Elimination Order (PEO)* is an EO such that v_i is simplical in G_i, i.e., $N_i(v_i)$ is a clique. An equivalent definition of a chordal graph is any graph that has a PEO. A PEO [29] and an optimal color assignment [13] can be computed in $O(|V| + |E|)$ time for chordal graphs.

B. Characterizing the IIG

Here, we derive structural properties that allow us to characterize an IIG for an application in SSA-LLP Form. These properties will be used in the following subsection to prove that the IIG is chordal. Let the set of procedures be $P = \{P_1, \ldots, P_k\}$ and $T = \{T_1, \ldots, T_M\}$ be the set of global registers. For each procedure P_i, let $G_i = (V_i, E_i)$ be its local interference graph. In Lemma 1 and Corollary 1, which follow, P_i and P_j are distinct procedures, i.e., $i \neq j$, in SSA-LLP Form.

Lemma 1. No variable v_i defined locally in P_i interferes with a variable v_j defined locally in P_j.

Corollary 1. No variable defined in procedure $P_1 = main$ can be involved in a global interference in an SSA-LLP form application.

In an IIG, the global registers are $T = \{T_1, \ldots, T_M\}$. $G_T = (T, E_T)$ is the induced subgraph of the IIG containing variables in T. Lemma 2 follows from the fact that $\delta_{max} = M$, and that the variables involved in global interferences are stored in T.

Lemma 2. $T = \{T_1, \ldots, T_M\}$ forms a clique in the IIG.

Corollary 2. Any EO $\sigma(T)$ is a PEO of G_T.

When P_i is called, $m = \delta_i$ variables reside in $T_1, \ldots, T_m$. Each global register T_j, where $1 \leq j \leq m$, interferes globally with every variable defined locally in P_i, e.g., the set V_i; likewise, no global register, T_j, where $m+1 \leq j \leq M$, interferes with any variable in V_i. The set of global interferences involving local variables in P_i is denoted $E_{(T,i)}$.

The IIG, $G^* = (V^*, E^*)$ is defined as follows:

$$V^* = T \cup \bigcup_{i=1}^{k} V_i \text{ , and} \quad (4)$$

$$E^* = E_T \cup \bigcup_{i=1}^{k} E_i \cup E_{(T,i)} \text{ .} \quad (5)$$

Fig. 3 shows the IIG for the application shown in Fig. 1 with the $|L(c_i)|$ and δ_i values taken from Table I. Local interferences are not shown. Since the application is in SSA-LLP Form, each subgraph G_i is a chordal graph. Many of the clique edges in E_T are not shown in order to make the illustration easier to follow.

C. The IIG is Chordal

We prove that an IIG for an application in SSA-LLP Form is chordal. Lemma 3 describes how to build an IIG for a procedure that includes a global interference at the call point. Corollary 3 then generalizes this result to multiple interferences.

Lemma 3. Let P be an SSA Form procedure with chordal interference graph $G = (V, E)$ and let v be a variable that is live across a call to P. Then the graph $G' = (V', E')$ induced by $V' = V \cup \{v\}$ is chordal.

Proof. Consider vertex $v_i \in V$. G has a PEO since it is chordal. v_i is a simplical vertex in subgraph $G_i = (V_i, E_i)$ induced by $V_i = \{v_1, \ldots, v_i\}$. In other words, $N_i(v_i)$ is a clique.

Now, let $V_i' = V_i \cup \{v\}$ and let $N_i'(v_i)$ be the set of neighbors of v_i in V_i'. If $V_0' = \{v\}$, v is simplical in V_0'. Since v interferes with every variable in V, it follows that $N_i'(v_i) = N_i(v_i) \cup \{v\}$ is a clique for $i > 0$. Therefore v_i is simplical in G_i'. □

Corollary 3. Let T be a set of variables that are live across a call to procedure P with chordal interference graph $G = (V, E)$. Then the subgraph $G' = (V', E')$ induced by $V' = V \cup T$ is chordal.

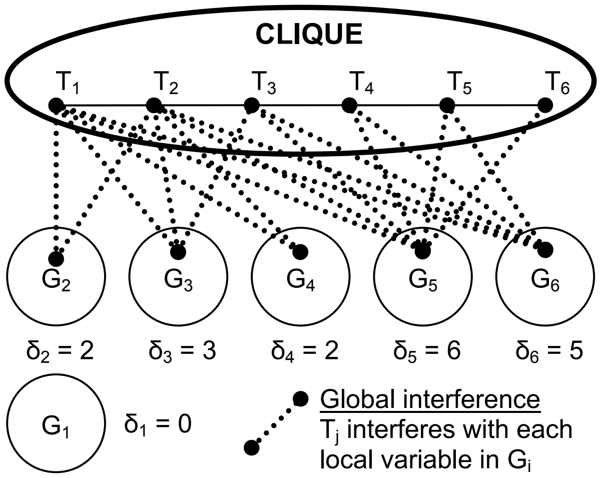


Figure 3. The IIG for the application depicted in Fig. 1 and Table I.

Let $\circ$: $\alpha \times \alpha \rightarrow \alpha$, be an operator that concatenates two EOs; e.g., $\alpha(X) \circ \alpha(Y) = \alpha(X)\alpha(Y) = \alpha(XY)$, where XY is the union of vertex sets (or subgraphs) X and Y, including all edges connecting a vertex in X to a vertex in Y. When implicit, e.g., $\alpha(X)\alpha(Y)$, $\circ$ may be omitted. Let $\alpha(G^*) = \sigma(T)\sigma(G_1)\sigma(G_2) \ldots \sigma(G_k)$ be an EO of G^*. $\sigma(T)$ and each term $\sigma(G_i)$ is a PEO. The composition of multiple PEOs of several subgraphs does not guarantee that $\alpha(G^*)$ is a PEO.

Consider a vertex $v \in V^*$. Let $N^*(v)$ be the set of vertices adjacent to v in G^*. If $\alpha(v) = i$, then let $N_i^*(v)$ be the set of vertices adjacent to v that precede v in α.

Theorem 1. $\alpha(G^*)$ is a PEO of G^*.

Proof. Assume to the contrary that $\alpha(G^*)$ is not a PEO of G^*. Then there is a vertex $v_i \in V$ such that $N_i^*(v_i)$ is not a clique.

First, consider the possibility that $v_i = T_i \in T$, i.e. v_i is a global register. By construction, the vertices in T precede all other vertices in α. So $N_i^*(T_i) = \{T_1, \ldots, T_{i-1}\}$. By Lemma 2, $N_i^*(T_i)$ is a clique. Thus, v_i cannot be a global register.

Therefore, v_i must be a local variable inside some procedure, P_j, with local interference graph $G_j = (V_j, E_j)$; i.e., $v_i \in V_j$. G_j is chordal since P_j is in SSA Form. Let U_i be the subset of vertices of V_j that precede v_i in $\sigma(G_j)$ and thus precede v_i in $\alpha(G^*)$. Since G_j is chordal and $\sigma(G_j)$ is a PEO of G_j, it follows that v_i is simplical in the subgraph of G_j induced by U_i.

Let $x, y \in N_i^*(v_i)$ be two non-adjacent vertices.

(1) By the reasoning above, both x and y cannot belong to U_i. This would contradict the fact that $\sigma(G_i)$ is a PEO of G_i.

(2) Neither x nor y can be defined locally in some procedure other than G_j. Since they interfere with v_i, which is defined locally in G_j, this would contradict Lemma 1, which states that two variables defined locally in different procedures cannot interfere.

(3) Both x and y cannot belong to T. Since they do not interfere, this would contradict Lemma 2 which states that T is a clique.

(4) By (1)-(3), without loss of generality, it follows that $x \in T$ and $y \in V_i$. Since $x \in N_i(v)$ and $x \in T$, the interference between x and v_i is global. Therefore x is live across the call to procedure P_j. By Corollary 3, x interferes with every variable defined locally in G_j, which includes y, contradicting the fact that x and y do not interfere.

Therefore $\alpha(G^*)$ is a PEO of G^*. □

Corollary 4. G^* is a chordal graph.

Henceforth, $\alpha(G^*)$ will be replaced with $\sigma(G^*)$ since $\alpha(G^*)$ is a PEO of G^*. σ represents a PEO, whereas α represents any EO.

D. Coloring the IIG

Here, we present an efficient optimal algorithm to color the IIG and we derive its time complexity. Like the algorithm of Beidas and Zhu [1], we do not construct the complete IIG. $\sigma(G^*)$ can be constructed deterministically, as long as we have pre-computed PEOs of each individual procedure and $\sigma(T)$; we focus solely on using this specific PEO. Let $R = max\{\delta_1 + \chi_1, \ldots, \delta_k + \chi_k\}$, where χ_i is the chromatic number of interference graph G_i for procedure P_i. Let $\chi(G^*)$ be the chromatic number of G^*, the IIG.

Theorem 2. $\chi(G^*) = R$.

Proof. Without loss of generality, let $R = \delta_i + \chi_i$. Since there is a sequence of calls from P_1 to P_i along which δ_i global variables are defined across the call to P_i, $R \geq \delta_i + \chi_i$. Hence, δ_i variables already reside in registers before calling P_i and at most χ_i variables are simultaneously live in P_i. Therefore $\chi(G^*) \geq R$.

Let $\omega(G^*)$ be the cardinality of the largest clique in G^*. Since all chordal graphs are *perfect graphs* [10], $\chi(G^*) = \omega(G^*)$. We must show that no clique C exists in G^* such that $|C| > R$. Now, assume to the contrary that some clique C does exist in G^* such that $|C| > R$.

Let $V(\delta_j + \chi_j)$ be a subset of vertices of G^* containing the first δ_j vertices in T (i.e., if $m = \delta_j$, then $T_1 \ldots T_m$ are the first m vertices) and a subset of χ_j vertices in V_i that form a maximal clique in G_j. $V(\delta_j + \chi_j)$ is a clique by Lemma 3 and Corollary 3. Since $R = \delta_i + \chi_i$, then every clique $V(\delta_j + \chi_j)$ must satisfy $|V(\delta_j + \chi_j)| \leq$ R; the contrary would contradict the fact that R is maximal taken across every procedure.

Since $C > R$, it follows that C must have some extra vertices from somewhere. There are two possible locations for extra vertices. If C includes vertices from $T_{m+1} \ldots T_n$, $n = |T|$, then C is not a clique because none of these vertices interfere with any variables defined locally in P_i. Therefore, the extra vertices must come from some procedure P_j, $j \neq i$. Since C is a clique, these extra variables must interfere with the variables defined locally in P_i, which contradicts Lemma 1. Therefore, $|C| = \chi(G^*) \leq R$. Since we have already shown that $\chi(G^*) \geq R$, it follows that $\chi(G^*) = R$. □

Now that we have established that G* is chordal and identified its chromatic number, we focus on coloring it optimally. For vertex $v \in V_i$, let *color(v)* be the color assigned to v when G_i is colored optimally and *color*(v)* be the color assigned to v when G^* is colored optimally. In other words, *color(v)* is the color assigned to v if G_i was colored separately, outside of the context of interprocedural register allocation. *color*(v)* is a color that could be assigned to v by optimally coloring the complete *IIG*. By relating *color*(v)* to *color(v)*, Theorem 3, which follows, effectively describes an optimal algorithm for coloring the IIG that simply colors each procedure individually. There is no need to construct the complete IIG, which makes this algorithm scalable like the CPP heuristic of Beidas and Zhu [1].

Theorem 3. $color^*(v) = color(v) + \delta_i$ is a legal color assignment for each variable v defined locally in procedure P_i.

Proof. For each $T_i \in T$, let $color(T_i) = i$. Since T is a clique (Lemma 2), $|T|$ colors are needed to color T.

Now, consider procedure P_i with interference graph $G_i = (V_i, E_i)$. Let $v \in V_i$ and let $\sigma(v) = j$, i.e., v is the j^{th} vertex in the PEO for G_i. The proof is achieved using induction on j.

If $j = 0$, then $color(v) = 1$ by Gavril's algorithm [13]. In G^*, $N_j^*(v) = \{T_1, \ldots, T_m\}$, where $m = \delta_i$, by Lemma 2 and Corollaries 2 and 3. Therefore the first available color for v is $1 + \delta_i$. Thus, $color^*(v) = color(v) + \delta_i$. For the induction, suppose that for $j < k$, every vertex v such that $\sigma(v) = j$ satisfies $color^*(v) = color(v) + \delta_i$. Now let v be the vertex in V_i such that $\sigma(v) = k$. If $N_i(v)$ is empty, then $color(v) = 1$ using the same reasoning as the basis, and $color^*(v) = 1 + \delta_i = color(v) + \delta_i$. Otherwise, for each color c, $1 \leq c < color(v)$, there must some vertex $u \in N_i(v)$ such that $color(u) = c$. Since $\sigma(v) < k$, it follows that $color^*(u) = c + \delta_i = color(u) + \delta_i$. Therefore colors $m+1 \ldots c$ are not available for v. Since $\{T_1, \ldots, T_m\} \in N_i^*(v)$, it follows that colors $1..m$ are not available for v either. Therefore the first color available for v is $color^*(v) = color(v) + \delta_i$. □

By Theorem 3, G^* can be colored optimally by first assigning colors $1..|T|$ to the vertices in T, and then coloring the chordal interference graph for each procedure P_i using the standard algorithm for chordal coloring, but with δ_i as an offset; G^* is never built.

E. Comparison to Top-Down Color Palette Propagation

It is important to acknowledge that the optimal method proposed in this paper can be viewed as a special case of the top-down CPP heuristic proposed by Beidas and Zhu [1]. Their approach made no assumptions regarding the program representation or which algorithm or heuristic is used to color the interference graph. As illustrated by Fig. 2, however, this approach could not achieve optimality without launch and landing pads.

Beidas and Zhu represented each procedure as a (non-SSA) control flow graph and colored the interference graph using Chaitin's heuristic [8]; in actuality, the heuristic used in Chaitin's register allocator was introduced in 1879 by Kempe [18]. This specific choice of program representation and heuristic cannot guarantee optimality.

F. Time Complexity

Theorem 4 states the time complexity of coloring the IIG as described in the proof of Theorem 3.

Theorem 4. The time complexity, $S(G^*)$, of coloring G^* is

$$S(G^*) = O\left[|T| + \sum_{i=1}^{k} (|V_i| + |E_i|)\right]. \qquad (6)$$

Proof. The time to assign colors $1..|T|$ to each variable in $|T|$ is $O(|T|)$. The time to apply chordal coloring to the interference graph G_i for procedure P_i is $O(|V_i| + |E_i|)$. □

The complexity of coloring the complete IIG using Gavril's algorithm is $S'(G^*) = O(|V^*| + |E^*|)$. $S'(G^*)$ includes two extra terms: $|E_T| = ½|T|(|T|-1) = O(|T|^2)$ and $|E_{(T,i)}| = \delta_i|V_i|$. Thus:

$$S'(G^*) = O\left[|T|^2 + \sum_{i=1}^{k} (\delta_i|V_i| + |E_i|)\right]. \qquad (7)$$

G. Further Discussion

The time complexity described above only describes the cost of computing a coloring. There are four terms whose contributions have not been taken into account for brevity.

The first omitted term is the complexity of computing the CPG. This requires a linear traversal of the instructions in each procedure in order to find the call points. When P_i calls P_j, one vertex (a call point c_k) and two edges, (P_i, c_k) and (c_k, P_j) are added to the CPG. If there are k procedures, the cost of finding P_j in a list is $O(k)$. If I is the total number of instructions in the application (across all procedures) and there are C call points, the time complexity becomes $O(|I| + |C|k)$.

The cost of looking up P_j can be reduced to near-constant by using a hash table. In the worst case, all procedures hash to the same bucket and the cost per-lookup is still $O(k)$. In the average-case, this cost can be mitigated by using a good hash function and allocating a table with a sufficient number of buckets.

The second and third omitted terms are the cost of computing the SCCs of the CPG to eliminate recursive procedure calls and the cost of computing the δ_i-values; both are $O(|V_{CPG}| + |E_{CPG}|)$.

The fourth and final omitted term is the cost of performing liveness analysis and building the interference graph for each procedure. The algorithms used for these procedures can be found in any compiler textbook (e.g. [11]). It is well-known that liveness analysis, in particular, is quite costly, but necessary, in practice.

VI. Experimental Results

We implemented the optimal interprocedural register allocation algorithm into the Machine SUIF compiler framework [26] and compared our results to the *color palette propagation (CPP)* heuristic of Beidas and Zhu [1]. The techniques described in this paper were all implemented; however, we do not support the extensions for function pointers, as described in Subsection III.C.

Beidas and Zhu described two different approaches to CPP: top-down, and bottom-up. In their experiments, they reported that bottom-up CPP generally reports better results than top-down; our experiments confirm this observation as well. It is thus quite interesting to note that the optimal algorithm described in this paper is a special case of the top-down approach, rather than bottom-up.

As discussed in Section V.E, Beidas and Zhu implemented a greedy coloring heuristic due to Kempe [18] that was used by Chaitin [8] in his register allocator. In our implementation, we opted for the *smallest last ordering (SLO)* heuristic of Matula and Beck [22] instead. The SLO heuristic later became the basis for the optimistic allocator of Briggs et al. [6], whose framework subsequently has been enhanced and improved by many others in the years since.

In the SLO heuristic, an EO is computed by repeatedly removing the vertex of smallest degree from the graph, and pushing it onto a stack. The reverse order of the stack is then used as the EO. Color assignment proceeds in the same fashion as Gavril's algorithm [13].

Using counting sort [25], the vertices of the graph can be sorted in increasing order of degree in $O(|V|)$ time and space. Consequently, the construction of the SLO and color assignment also has a time complexity of $O(|V| + |E|)$, the same as chordal graph coloring.

One important difference, however, is that an optimal color assignment for an SSA Form procedure can be computed without constructing an interference graph [7, 17]. SLO, however, requires an interference graph, the construction of which can be quite slow. In practice, this gives the optimal algorithm a significant runtime advantage over the sub-optimal heuristics.

We selected open-source benchmarks from the Mediabench [21] and MiBench [15] suites of embedded and multimedia applications. Tables II and III show the results of our experiments. Table II shows the number of registers allocated using: (1) Top-Down CPP *(CPP↓)*, (2) Bottom-Up CPP *(CPP↑)*, and (3) the optimal algorithm *(Opt)*; optimal solutions are shown in bold and underlined. Table III shows the runtime of the different approaches. 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.

From Table II, we can see that in all cases, *CPP*↑ performed significantly better than *CPP*↓ for all benchmarks, meaning that it allocated fewer registers. In many cases, *CPP*↑ allocated the same number of registers as *Opt*; of course, since *Opt* is optimal, *CPP*↑ could not feasibly allocate fewer registers. From Table III, we can see that *Opt* ran significantly faster than both *CPP*↓ and *CPP*↑. The runtimes reported in Table III do not account for liveness analysis, but they do include the cost of interference graph construction in the case of *CPP*↓ and *CPP*↑; *Opt*, as stated above, computes a coloring without constructing an interference graph. The greatest disparity in runtime was for the benchmark *pegwit*, which had the largest overall runtime. For *pegwit*, *CPP*↓ and *CPP*↑ respectively ran 149× and 153× slower that *Opt* respectively.

To summarize, *Opt* runs significantly faster than the *CPP*↓ and *CPP*↑ heuristics. These heuristics, in turn, were shown by Beidas and Zhu [1] to run more than 100× faster than the preceding algorithm by Vemuri et al. [31].

TABLE II. Number of Registers Allocated by the 2 CPP Heuristics and the Optimal Algorithm. Optimal solutions are shown in bold and underlined

Benchmark	CPP ↓	CPP ↑	Opt
adpcm_rawcaudio	21	**<u>15</u>**	**<u>15</u>**
adpcm_rawdaudio	23	**<u>15</u>**	**<u>15</u>**
blowfish	36	**<u>20</u>**	**<u>20</u>**
crc32	13	12	**<u>11</u>**
dijkstra_large	14	10	**<u>9</u>**
dijkstra_small	14	10	**<u>9</u>**
fft	31	21	**<u>20</u>**
g721_decoder	39	26	**<u>25</u>**
g721_encoder	38	23	**<u>22</u>**
gsm	38	**<u>31</u>**	**<u>31</u>**
mpeg2dec	70	48	**<u>47</u>**
mpeg2enc	105	94	**<u>91</u>**
patricia	22	**<u>13</u>**	**<u>13</u>**
pegwit	38	33	**<u>32</u>**
sha	23	**<u>18</u>**	**<u>18</u>**
susan	39	**<u>23</u>**	**<u>23</u>**

TABLE III. Runtime (milliseconds, 3 Significant Digits of Accuracy) of the 2 CPP Heuristics and the Optimal Algorithm.

Benchmark	CPP ↓	CPP ↑	Opt
adpcm_rawcaudio	119	98.8	51.1
adpcm_rawdaudio	74.3	62.6	53.8
blowfish	4500	3980	148
crc32	48.1	38.2	10.1
dijkstra_large	90.5	66.3	65.7
dijkstra_small	86.8	59.5	46.8
fft	348	301	95.3
g721_decoder	1550	853	158
g721_encoder	2140	877	145
gsm	5830	5410	636
mpeg2dec	5040	4320	824
mpeg2enc	14200	13000	1560
patricia	308	256	123
pegwit	166000	170000	1110
sha	283	245	97.3
susan	41200	40100	1140

The runtimes reported in Table III only correspond to the cost of computing a coloring of the interference graph. For example, the cost of converting each procedure to SSA Form, in the case of *Opt*, is not accounted for. On the other hand, most compilers do use SSA Form to perform optimizations such as *constant propagation* and *dead code elimination*. In this case, there is no reason to translate out of SSA Form and then rebuild it prior to running *Opt*; moreover, the cost of translating out of SSA Form should be added to the runtimes of *CPP*↓ and *CPP*↑; on the other hand, in the case of ASIP design, the translation out of SSA Form occurs after register allocation and assignment. In conclusion, the real runtime overhead of register allocation, when different intermediate representations are compared, depends on the context in which register allocation is applied.

VII. Conclusion and Future Work

This paper has presented an optimal algorithm for interprocedural register allocation in the context of high-level synthesis and automated ASIP design. The proposed method augments the traditional SSA Form of procedures with launch and landing pads, which are parallel copy operations placed immediately before and after each procedure call. The launch and landing pads copy values to a set of global registers that have been pre-allocated in advance. The global registers hold values that are live across procedure calls, and are allocated by considering the maximum number of variables that are live across all non-recursive paths in the call graph.

At the core of the optimal algorithm is a proof that an IIG constructed for a program in SSA Form is a chordal graph. Fortunately, the new method retains the scalability of the CPP heuristics proposed by Beidas and Zhu [1]; in other words, an optimal allocation can be achieved by coloring each procedure individually, and there is no need to construct the complete IIG. Our experiments show that the optimal algorithm runs significantly faster than previous heuristics, primarily due to the fact that register can be allocated to an SSA Form procedure without constructing an interference graph; for a proof of this fact, refer to prior work by Brisk et al. [7] and Hack et al. [17].

There are several areas for future work that can extend this paper. The first is to consider the possibility of register assignment, both in the context of high-level synthesis and ASIP design. In the context of synthesis, register assignment seeks to minimize interconnect costs, and as such, is part of a much larger NP-Complete connectivity binding problem [24]. In the context of ASIP design, the goal is to find a color assignment that minimizes the number of copies dynamically executed [14]; this problem is also NP-Complete [3]. Lastly, as stated in the introduction, the proposed method does not account for the lifetimes of static variables. To account for static variables, an algorithm for interprocedural liveness analysis for them is required; we intend to develop such an algorithm in the future.

REFERENCES

[1] R. Beidas and J. Zhu, "Scalable interprocedural register allocation for high-level synthesis," Asia South Pacific Design Automation Conf., Shanghai, China, pp. 511-516, January 2005.

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

[3] F. Bouchez, A. Darte, and F. Rastello, "On the compleixty of register coalescing" Int. Symp. Code Generation and Optimization, San Jose, CA, USA, pp. 102-114, March 2007.

[4] F. Bouchez, A. Darte, and F. Rastello, "On the compleixty of spill everywhere under SSA form" Conf. Languages, Compilers, and Tools for Embedded Systems, San Diego, CA, USA, pp. 103-112, June 2007.

[5] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, "Practical improvements to the construction and destruction of static single assignment form," Software—Practice and Experience, vol. 28, no. 8, pp. 859-881, July 1998.

[6] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," ACM Trans. Programming Languages and Systems, vol. 16, no. 3, pp. 428-455, May 1994.

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

[8] G. J. Chaitin, "Register allocaiton and spilling via graph coloring," SIGPLAN Symp. Compiler Construction, Boston, MA, USA, pp. 98-101, June 1982.

[9] F. Chow, "Minimizing register usage penalty at procedure calls," Int. Conf. Programming Language Design and Implementation, Atlanta, GA, USA, pp. 85-94, June 1988.

[10] M. Chudnovsky, N. Robertson, P. Seymour, and R. Thomas, "The strong perfect graph theorem," Annals of Mathematics, vol. 164, no. 1, pp. 51-229, July, 2006.

[11] K. D. Cooper and L. Torczon, Engineering a Compiler, San Francisco: Morgan-Kaufmann, 2003.

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

[13] F. Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph," SIAM J. Comput., vol. 1, no. 2, pp. 180-187, June 1972.

[14] D. Grund and S. Hack, "A fast cutting-plane algorithm for optimal coalescing," Int. Conf. Compiler Construction, Lisbon, Portugal, pp. 111-125, March 2007.

[15] M. R. Guthaus, et al., "MiBench: a free commercially representative embedded benchmark suite," Workshop on Workload Characterziation, Austin, TX, USA, pp. 3-14, December 2001.

[16] S. Hack and G. Goos, "Optimal register allocation for SSA-form programs in polynomial time," Information Processing Letters, vol. 98, no. 4, pp. 150-155, May 2006.

[17] S. Hack, D. Grund, and G. Goos, "Register allocation for programs in SSA-form," Int. Conf. Compiler Construction, Vienna, Austria, pp. 247-262, March 2006.

[18] A. B. Kempe, "On the geographical problem of the four colors," American Journal of Mathematics, vol. 2, pp. 193-200, 1879.

[19] F. J. Kurdahi and A. C. Parker, "REAL: a porgrma for register allocation," Design Automation Conf. Miami Beach, FL, USA, pp. 210-215, June-July 1987.

[20] S. M. Kurlander and C. N. Fischer, "Minimum cost interprocedural register allocation," Symp. Principles of Programming Languages, St. Petersburg Beach, FL, USA, pp. 230-241, January 1996.

[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Int. Symp. Microarchitecture, Research Triangle Park, NC, USA, pp. 330-335, December 1997.

[22] D. W. Matula and L. L. Beck, "Smallest last-ordering and clustering graph coloring algorithms," Journal of the ACM, vol. 30, no. 3, pp. 417-427, July 1983.

[23] R. Muth and S. K. Debray, "On the comlexity of function pointer may-alias analysis," Int. Joint Conf. CAAP/FASE on Theory and Practice of Software Development, Lille, France, pp. 381-392, April 1997.

[24] B. Pangrle, "On the complexity of connectivity binding," IEEE Trans. Computer Aided Design, vol. 10, no. 11, pp. 1460-1465, November 1991.

[25] H. H. Seward, "Information sorting in the application of electronic digital copmuters to business operations," M.S. Thesis, Masschusetts Institute of Technology, 1954.

[26] M. D. Smith and G. Holloway, "An introduction to Machine SUIF and its portable libraries for analysis and optimization," Technical Report, Harvard University, 2002. Available online.

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

[28] R. E. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Comput., vol. 1, no. 2, pp. 146-160, June 1972.

[29] R. E. Tarjan and M. Yannakakis, "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs," SIAM J. Comput., vol. 13, no. 3, pp. 566-579, August 1984.

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

[31] R. Vemuri, S. Katkoori, M. Kaul, and J. Roy, "An efficient register optimization algorithm for high-level synthesis from hierarhical behavioral specifications," ACM Trans. Design Automation of Electronic Systems, vol. 7, no. 1, pp. 189-216, January 2002.

[32] S. Zhang and B. G. Ryder, "Complexity of single level function pointer aliasing analysis," Tech. Report LCSR-TR-233, Laboratory of Computer Science Research, Rutgers University, October 1994.