Optimal Polynomial-Time Interprocedural Register Allocation for High-Level Synthesis Using SSA Form

Philip Brisk Ajay K. Verma Paolo Ienne
Processor Architecture Laboratory
Swiss Federal Institute of Technology, Lausanne (EPFL)
Lausanne, Switzerland
{philip.brisk, ajaykumar.verma, paolo.ienne}@epfl.ch

ABSTRACT
An optimal, polynomial-time algorithm for interprocedural register allocation in high-level synthesis and ASIP design is presented. The algorithm determines the minimum number of registers required to store all scalar variables in an application without spilling any to memory. Although an optimal polynomial-time algorithm has been presented in the past for individual procedures in Static Single Assignment (SSA) Form, this is the first such claim for the interprocedural analogue of the problem, which considers interferences across procedure calls. A new extension of SSA Form is introduced, and an example illustrates that this new representation can reduce the number of registers required for an optimal allocation. A secondary aspect of the optimal algorithm is that it is scalable: there is no need to build a complete interprocedural interference graph and each procedure can be colored individually. Our experiments show that the optimal algorithm runs more than 100× faster than a previously published scalable sub-optimal heuristic for the same problem.

Categories and Subject Descriptors
B.5.2 [**Hardware**]: Register-Transfer Level Implementation – *automatic synthesis; optimization*.

General Terms
Algorithms, Performance, Design.

Keywords
High-Level Synthesis, Register Allocation, SSA Form.

1. INTRODUCTION
Register allocation in the context of *high-level synthesis (HLS)* and *application-specific (instruction-set) processor (ASIP)* design is the problem of determining precisely how many registers are required for the system. Typically, 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 *x* and *y* overlap. Two such *interfering* variables cannot share the same storage location.

A subset $S \subseteq V$ of variables can share the same register if and only if *S* is an *independent set*, i.e. the vertices in *S* are pairwise non-adjacent. The goal of the *minimum coloring problem* is to partition *V* into the minimum number of non-overlapping independent sets. In the context of coloring, all of the vertices belonging to the same independent set are assigned an integer value, called a *color*, and each independent set is called a *color class*.

If there are *k* independent sets, then colors *1..k* are assigned to the vertices in each color class. A register is allocated to the system for each color class, and each variable assigned to the i^{th} color assign is then bound to the i^{th} register.

Although graph coloring is NP-Complete, there are many classes of graphs for which the coloring problem can be solved optimally in polynomial-time. In particular, this paper focuses on the class of *chordal graphs*, which can be colored optimally in $O(|V| + |E|)$ time using a greedy algorithm by Gavril [10]. The method proposed in this paper ensures that the interprocedural interference graph is a chordal graph, and a method is presented that colors each procedure optimally and individually.

The key to the success of the algorithm presented in this paper is a novel interprocedural program representation that extends *Static Single Assignment (SSA) Form*; the new representation is called *SSA Form with Launch and Landing Pads (SSA-LLP)*. SSA-LLP Form pre-allocates the minimum number of caller-save registers to the design to hold all of the variables that are live across every procedure call point in the program; by copying variables to and from these registers before and after each call, SSA-LLP form ensures that no variables defined locally in two separate procedures interfere with one another, while ensuring that the interprocedural interference graph is chordal.

The color assignment procedure outlined here is scalable. It does not require the construction of a complete interprocedural interference graph, and the interference graph for each procedure can be colored individually. It runs more than 100× faster than a previously-published scalable sub-optimal heuristic [1].

2. RELATED WORK
2.1 Register Allocation
Techniques for register allocation in high-level synthesis have developed over the years as the granularity of the application being synthesized has increased. In the mid-1980s, the typical application was a *dataflow graph (DFG)* whose operations had already been scheduled and bound to resources. A DFG can represent an acyclic program—via if-conversion, it can handle conditions, but it cannot represent loops. In 1986, Tseng and Siewiorek [26] formulated register allocation as a clique partitioning problem on a compatibility graph, the inverse of an interference graph; a sub-optimal heuristic was presented. In 1987, Kurdahi and Parker [17] showed that the interference graph for a DFG was an interval graph that could be colored optimally using the *Left Edge Algorithm* [28], originally developed by Hashimoto and Stevens [14] for channel routing. Springer and Thomas [21] showed that interference graphs are chordal if restrictions are placed on variable lifetimes and procedure calls.

A *cyclic DFG* has feedback edges and can thus represent loops. Stok [23] showed that an interference graph for a scheduled cyclic DFG is a circular arc graph, for which coloring is NP-Complete.

In 2005 and 2006, Bouchez et al. [2], Brisk et al. [5], and Hack and Goos [12] independently proved that an interference graph for a program represented in SSA Form is a chordal graph. In short, SSA Form imposes the same restrictions on variable lifetimes that were noted by Springer and Thomas a decade earlier.

Vemuri et al. [27] were the first to study interprocedural register allocation for high-level synthesis. They built an *interprocedural interference graph (IIG)*, which includes both local interferences as well as global interferences across procedure calls. Beidas and Zhu [1] developed a scalable algorithm using a technique called *Color Palette Propagation (CPP)* that avoided building a complete IIG and colored each procedure individually. Top-down and bottom-up CPP techniques propagated interferences across procedure call boundaries. Their results were comparable to Vemuri et al. with a runtime that was 100× faster. Both of these techniques are heuristics that cannot claim optimality.

There has been considerable work on register allocation in compilers (e.g. [4, 6])—far too much to enumerate here. Since the number of registers in the target architecture is fixed, the primary goal of such allocators is to minimize the cost of spilling variables to memory; a secondary goal is to assign registers to eliminate as many copies as possible. Computing a minimal coloring of an interference graph does not suffice to solve these problems.

Relevant to this work, however, is interprocedural register allocation in compilers [7, 22]. Many RISC architectures dedicate certain registers as caller-save and callee-save, and the goal of such allocators is to minimize the cost of saving and restoring variables at procedure call and return points. The SSA-LLP representation effectively preallocates a sufficient number of caller-save registers to handle all the variables that are live across each call point in the application, including every possible transitive sequence of calls. This will be described in greater detail in Section 4.

2.2 Chordal Graphs

Let $G = (V, E)$ be an undirected graph. A cycle is a set of vertices $\{v_0, v_1, ..., v_j\}$ in the graph, such that there is an edge $(v_i, v_{(i+1) \bmod j})$ for $i = 0, 1, ..., j$. A *chord* is an edge (v_s, v_t) that is not part of the cycle, i.e. $t \neq (s+1) \bmod j$. A *chordless cycle (hole)* [8] is a cycle of length at least 4 that does not contain a chord (note that a cycle of length 3 is a clique, and a cycle of length 2 is simply an edge between a pair of vertices). A *chordal graph* is defined to be any graph that contains no chordless cycles of length 4 or more.

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 set $\{1, .., |V|\}$ to each vertex. Given an EO, vertices are named such that $\sigma(v_i) = i$. Let $V_i = \{v_j \mid j \leq i\}$, and $G_i = (V_i, E_i)$ be the subgraph of G induced by V_i; G_0 is the empty graph, and $N_i(v_j) = \{v_k \in N(v_j) \mid k < i\}$. In other words, $N_i(v_j)$ contains all vertices adjacent to v_j that occur before v_j in the EO.

In a graph G, vertex v is defined to be *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. that $N_i(v_i)$ is a clique, for each vertex $v_i \in V$. An equivalent definition of a chordal graph is any graph that has a PEO. A PEO [25] and an optimal color assignment [10] can both be computed in $O(|V| + |E|)$ time for chordal graphs.

2.3 Compiler Preliminaries

To conserve space, we assume that the reader is familiar with compiler concepts such as the *Control Flow Graph*, *Liveness Analysis*, and how to construct an interference graph; if not, a compiler textbook (e.g. [9]) should be consulted. Although this paper extends *Static Single Assignment (SSA) Form*, the reader does not need to understand the details; the paper by Briggs et al. [3] is probably the most accessible reference on SSA Form.

2.3.1 The Interprocedural Interference Graph

Two variables interfere if they are both *live* at some point in an application. *Local interferences* are between two variables in the same procedure, and they can easily be detected using liveness analysis [9]. *Global interferences* are interferences between variables across procedure calls. For example:

$$\begin{aligned} &X \leftarrow ... \\ &CALL\ A \\ &... \leftarrow X \end{aligned} \qquad (1)$$

For simplicity, assume that X is not a parameter passed to A and that there are no recursive function calls. X globally interferes with every variable defined locally in A; Beidas and Zhu [1] call these *immediate global conflicts*. X will also transitively interfere with all local variable defined in any procedure that could be called before A terminates. Beidas and Zhu call these transitive interferences *global conflicts*. We do not distinguish between immediate and non-immediate global conflicts.

An *Interprocedural Interference Graph (IIG)* 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 ever pair of variables x and y that interfere locally or globally. Any legal coloring of G is a legal solution to the interprocedural register allocation problem for synthesis.

2.3.2 The Call-Points Graph

Let P be the procedures in an program, and C be the set of points in the program where one procedure calls another. We assume that all calls are direct, i.e. there are no function pointers. This ensures that only one procedure is called from each call point.

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 point where P_i calls procedure P_j. Then edges (P_i, c_k) and (c_k, P_j) are added to E_{CPG}. This ensures that each call point has exact one predecessor (the caller) and one successor (the callee) in the CPG. 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 function(s). Clearly, no variable can reside in a register across a mutually recursive function call. Otherwise, the first recursive call to the same function will overwrite the variable's value. The only way to store variables across function calls is to push them onto a runtime stack, which is exactly what software compilers do. The same must be done for hardware synthesis; or alternatively, recursive function calls cannot be supported.

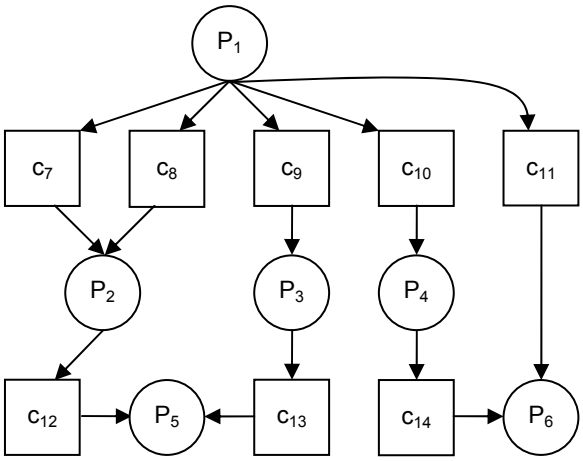

Figure 1. An example CPG.

For the purpose of register allocation, it suffices to eliminate recursive function calls from the CPG since locally defined variables will reside in memory across these call points. It suffices to compute the *strongly connected components (SCCs)* [24] of the CPG, and collapse each SCC into a single node. The result is called the *Augmented Components Graph* in graph theoretic literature. Throughout the remainder of this paper, we can thus assume that the CPG is acyclic to simplify the discussion.

3. GLOBAL INTERFERENCES

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 local 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

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

The total number of registers required to store variables along this particular path is $|L(T)|$. One approach to computing δ_j would be to enumerate every possible path from P_1 to P_j and select the largest *L-value* among all of these paths; however, there are an exponential number of unique paths in a DAG in the worst-case.

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, $\delta_k = \delta_i + |L(c_k)|$.

Table 1.

Example of the δ_i values for the CPG in Fig. 1 using the $|L(c_i)|$ values provided in the second column.

Call Point	$\lvert L(c_i)\rvert$	δ_i
c_7	1	1
c_8	2	2
c_9	3	3
c_{10}	2	2
c_{11}	5	5
c_{12}	3	5
c_{13}	3	6
c_{14}	2	4

Procedure	δ_i
P_1	0
P_2	2
P_3	3
P_4	2
P_5	6
P_6	5

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 1 above.

4. LAUNCH AND LANDING PADS

In this section, we present a new program representation that ensures that the IIG is a chordal graph. The initial step is to compute δ_i for each procedure P_i as described in the preceding section. Let $\delta_{max} = max\{\delta_1, \ldots, \delta_N\}$, $N = |V_{AC\text{-}CPG}|$. δ_{max} is the number of registers required to hold variables that are live across procedure calls and are involved in global interferences. A *leaf* is a procedure with no successors in the CPG. Clearly, δ_{max} corresponds to an δ_i-value for a leaf, although there may be non-leaf procedures whose δ_i-values are equal to δ_{max}.

Next, we allocate $M = \delta_{max}$ *global registers* $T = \{T_1, \ldots, T_M\}$ to hold these values. Now, 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 $T_{m+1}, \ldots, T_{m+n}$.

In general, we cannot assume that the color assignment phase can and/or will be able to assign these variables to the desired registers. To make this assignment feasible, we introduce parallel copy instructions—called *Launch* and *Landing Pads* 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. Both Ψ and Ψ^{-1} are parallel copy operations, similar in principle to φ-functions in SSA Form [12]. A procedure call augmented with launch and landing pads would have the following form:

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

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.

Any instruction of the form $y \leftarrow ...$ *defines* variable *y*. One of the defining features of SSA Form is that variables are defined exactly once [3]. *SSA Form with Launch and Landing Pads (SSA-LLP)* relaxes this constraint. For example, each variable in $L(c_k)$ is now defined multiple times: once at the original definition point, and now once by a landing pad. This is not problematic, however, as the LLP extension to SSA is only required for register allocation. The launch and landing pads do not need to be inserted prior to register allocation. Thus, any other SSA-based optimization or analysis can be applied without concern. Alternatively, the representation 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 use of the global registers in *T*.

There is a distinct similarity between the launch and landing pads and caller-save registers [7, 22] 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*.

4.1 Example

Fig. 2 shows an example that illustrates how SSA-LLP Form can reduce the chromatic number of an IIG. 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; 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* [8]. First and foremost, this graph is not chordal because it contains a chordless cycle. Second, 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). This interference graph is chordal; since all chordal graphs are *perfect graphs*, the chromatic number is equal to the maximal clique; here, both values are 2. Fig. 2 illustrates that converting from SSA Form to SSA-LLP Form can reduce the chromatic number of the IIG, and thus the number of registers allocated to the datapath (HLS) or ASIP register file.

5. CHARACTERIZING THE IIG

Lemmas 1 and 2 and Corollaries 1 and 2, which follow, allow us to characterize an IIG for an application in SSA-LLP Form. Let the set of procedures be $P = \{P_1, ..., P_k\}$ and $T = \{T_1, ..., T_M\}$ be the set of global registers. For each procedure P_i, let $G_i = (V_i, E_i)$ be its local interference graph. The 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; a few proofs are omitted to conserve space.

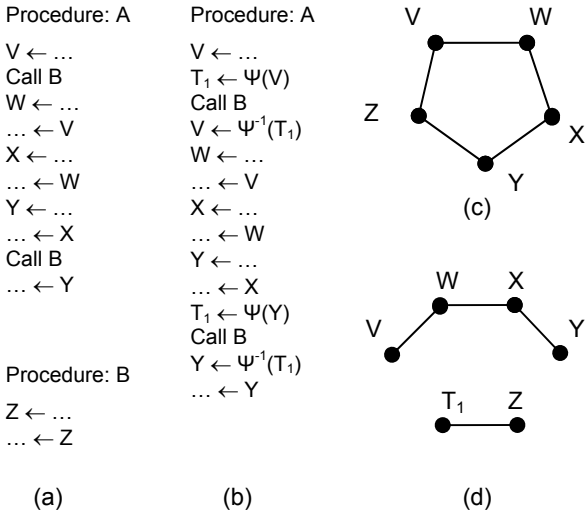

Figure 2. A small SSA-Form program (a) converted to SSA-LLP Form (b) and respective IIGs (c) and (d).

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 $T = \{T_1, ..., T_M\}$. $G_T = (T, E_T)$ 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. $T = \{T_1, ..., T_M\}$ forms a clique in the IIG.

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

When procedure P_i is called, $m = \delta_i$ variables reside in global registers $T_1, ..., T_m$, as discussed in Section 4. The purpose of the launch and landing pads is to ensure that these variables are assigned to this specific set of registers. 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:

$$V^* = T \cup \bigcup_{i=1}^{k} V_i \tag{4}$$

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

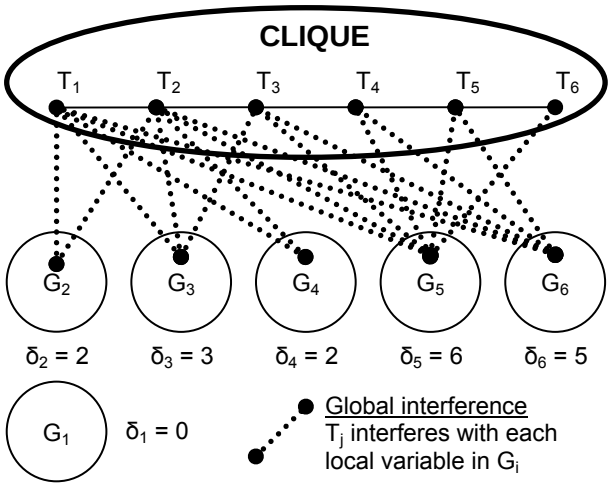

Figure 3. The IIG for the application depicted in Fig. 1 with $|L(c_i)|$ and δ_i values taken from Table 1.

Fig. 3 shows the IIG for the application shown in Fig. 1 with the $|L(c_i)|$ and δ_i values taken from Table 1. 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.

5.1 THE IIG is Chordal

Here, we prove that an IIG for an application in SSA-LLP Form is a chordal graph. Lemma 3 describes how to build an IIG for a procedure that includes a global interference at the call point.

Lemma 3. Consider a procedure P in SSA Form 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 $v_i \in V$. Since G is chordal G has a PEO. v_i is simplical in subgraph $G_i = (V_i, E_i)$ induced by $V_i = \{v_1, ..., v_i\}$. 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.

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 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 $\alpha(G^*) = \sigma(T)\sigma(G_1)\sigma(G_2)... \sigma(G_k)$ be an EO of G^*. $\sigma(T)$ and each $\sigma(G_i)$ term is a PEO. In the remainder of this section, we prove 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 $\alpha(G^*)$.

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 some vertex $v \in V$ such that $N_i^*(v)$ is not a clique.

If $v \in T$, let $v = T_j$. So $N_i^*(v) = \{T_1, ..., T_{j-1}\}$. Observe that all vertices in T precede all others in α by construction. Therefore the subgraph of G^* induced by $N_i^*(T_j)$ is not a clique, which contradicts Lemma 2.

Otherwise, let $v \in V_i$ for some interference graph $G_j = (V_j, E_j)$ that corresponds to procedure P_j. First, note that G_j is a chordal graph since P_j is a procedure in SSA Form. Let U_j be the subset of vertices of V_j that precede v in $\sigma(G_j)$ and thus precede v in $\alpha(G^*)$. Since G_j is chordal and $\sigma(G_j)$ is a PEO of G_j, it follows that v is simplical in the subgraph of G_j induced by U_j.

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

(1) By the reasoning above, both x and y cannot belong to U_j. 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, 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), it follows, without loss of generality, 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 is global. Therefore x is live across the call to procedure P_i. 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.

6. COLORING THE IIG

In this section, we present an efficient algorithm to color the IIG. We prove that the algorithm is optimal and derive its time complexity. Like the algorithm of Beidas and Zhu [1], we do not construct the complete IIG. This ensures that the algorithm is not only optimal, but practical.

Since $\sigma(G^*)$ can be constructed deterministically, as long as we have pre-computed PEOs of each individual procedure as well as $\sigma(T)$, we focus solely on using this specific PEO.

Let $R = max\{\delta_1 + \chi_1, ..., \delta_n + \chi_n\}$, 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 at least one path 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* [8], $\chi(G^*) = \omega(G^*)$. We must show that no clique C exists in G^* such that $|C| > R$. 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 the IIG is chordal, 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. 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 [10] 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 [10]. 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 $1 + \delta_i$. Therefore $color^*(v) = color(v) + \delta_i$.

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, the vertices in G^* can be colored 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. G^* is never built.

6.1 Time Complexity

We analyze the time complexity of coloring the IIG as described in the proof of Theorem 3; Theorem 4 states the result.

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

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

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: $|E_T| = \frac{1}{2}|T|(|T|-1) = O(|T|^2)$ and $|E_{(T,i)}| = \delta_i|V_i|$. Thus

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

6.1.1 Further Discussion

The time complexity described above only includes the cost of computing a coloring. There are several 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 last 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. [9]). It is well-known that liveness analysis, in particular, is quite costly in practice.

7. EXPERIMENTAL RESULTS

We implemented the optimal interprocedural register allocation algorithm into the Machine SUIF compiler framework [20] and compared our results to the *color palette propagation (CPP)* heuristic of Beidas and Zhu [1]. Beidas and Zhu described two different approaches to color propagation: top-down, and bottom-up. To color each procedure individually, they use Chaitin's heuristic [6], taken from register allocation in compilers; however, this heuristic actually dates back to the work of A. B. Kempe in 1879 [15]. An improvement to Kempe's heuristic was proposed in 1983 by Matula and Beck [19], and this heuristic later became the basis for the optimistic allocator developed by Briggs [4] (and subsequently enhanced and improved by many others).

We compare the optimal solution using SSA-LLP form presented here to both the top-down and bottom-up CPP approaches using Matula and Beck's coloring heuristic. For the CPP heuristic, we represented each procedure as a *Control Flow Graph (CFG)*, as was done by Beidas and Zhu [1]; for the optimal heuristic, we represented the complete application in SSA-LLP Form.

For our benchmarks, we selected a set of embedded applications from Mediabench [18] and MiBench [11]. The number of registers allocated is shown in Table 2 and the runtime is shown in Table 3. Since the algorithm presented here is optimal, the two heuristics can do no better.

From Table 2, we see that the bottom-up heuristic never allocates more registers than the top-down heuristic, and in many cases, it allocates significantly fewer. In many cases, the bottom-up heuristic does allocate the same number of registers as the optimal algorithm; however, this is purely coincidental.

It should be noted that the optimal algorithm can be viewed as a specific implementation of top-down propagation using SSA-LLP form; the only colors that are propagated downward are the global registers, which have been pre-allocated and pre-colored. The computation of δ_i-values does most of this work. For procedure P_i, colors *1* through δ_i are propagated; G_i is then colored optimally using Gavril's algorithm [10], but with δ_i+1 rather than *1* as the first available color.

Table 2.

Number of Registers Allocated.

Benchmark	Top Down	Bottom-Up	Optimal
adpcm_decoder	23	15	15
adpcm_encoder	22	15	15
blowfish	22	20	20
crc32	16	12	11
dijkstra_large	10	10	9
dijkstra_small	10	10	9
fft	22	21	20
g721_decoder	41	26	25
g721_encoder	32	22	22
gsm	35	31	31
mpeg2_decoder	62	48	47
mpeg2_encoder	112	94	91
patricia	28	13	13
pegwit	41	33	32
sha	21	18	18
susan	36	23	23

Table 3.

Runtime (seconds)

Benchmark	Top Down	Bottom-Up	Optimal
adpcm_decoder	0.136	0.125	0.00994
adpcm_encoder	0.177	0.153	0.047
blowfish	8.06	6.75	0.136
crc32	0.517	0.421	0.00518
dijkstra_large	0.113	0.0886	0.0408
dijkstra_small	0.113	0.088	0.0372
fft	0.486	0.446	0.507
g721_decoder	1.93	1.45	0.064
g721_encoder	2.8	1.47	0.0709
gsm	9.55	9.44	0.289
mpeg2_decoder	9.65	9.02	0.407
mpeg2_encoder	28.957	28.824	0.645
patricia	0.446	0.396	0.0531
pegwit	288.2	271.7	0.0585
sha	0.425	0.385	0.573
susan	66.222	63.51	0.324
Total	418	394	3.27
Average	26.1	24.6	0.204

The experiments were performed on a laptop PC with a 2.00 GHz Intel Pentium M processor with 1.00 Gigabytes of RAM running Fedora Linux. From Table 3, the optimal algorithm actually runs much faster than either the top-down or bottom-up heuristics. On average, we observed a speedup of 128× for the optimal algorithm compared to top-down CPP and 121× compared to bottom-up CPP. The complexity of the Matula and Beck heuristic is $O(|V|^2)$; whereas that of chordal coloring is $O(|V| + |E|)$. The complexity of both top-down and bottom-up CPP includes the cost of coloring each interference graph as well as the cost of propagating the palette; the interested reader should refer to the paper by Beidas and Zhu [1] for details.

As demonstrated by the experiments presented here, the optimal algorithm is significantly faster than either top-down or bottom-up CPP, which were the fastest heuristics for interprocedural register allocation in synthesis that have been published to date. Admittedly, the CPP heuristics are agnostic about how to represent each procedure (e.g. CFG vs. SSA Form) and which heuristic should be used to color each procedure individually. In principle, one might be able to do better than the results shown here for top-down and bottom-up CPP by representing each procedure in SSA Form (but not SSA-LLP) and coloring each procedure optimally using chordal coloring rather than the Matula and Beck heuristic. We have not done so, however, because the algorithm presented here is already optimal and fast, and thus, there is no need to explore further sub-optimal alternatives.

8. CONCLUSION AND FUTURE WORK

An optimal polynomial-time algorithm for interprocedural register allocation in the context of HLS and ASIP design has been presented. To the best of our knowledge, this is the first non-trivial interprocedural problem in HLS for which a polynomial-time solution has been found. Experimentally, we have found that the optimal algorithm is more than 100× faster than the top-down and bottom-up CPP heuristics of Beidas and Zhu [1], which are the state-of-the-art at the present.

We envision several avenues for future work. One extension involves integrating register assignment with operation and connectivity binding to minimize the cost of multiplexers and wires inserted into the design; this problem is NP-Complete, even for chordal interference graphs. Another possibility is to explore SSA-LLP form for interprocedural register allocation in compilers. Current work on interprocedural register allocation does not use SSA Form, and only one SSA-based allocator has been published to date [13].

The discussion in this paper has omitted a few aspects of the *C* language, such as function pointers. A function pointer allows a call point to call multiple functions. This can easily be handled by the technique presented in this paper. The problem, however, is that pointer analysis [15] is undecidable in the general case. We rewrote the *g721* and *gsm* applications so that we could compile them without function pointers. Due to the widespread use of function pointers, we were unable to compile *jpeg*.

This paper has not addressed *static* variables, which can reside in registers across recursive function calls. Static variables are defined only the first time that a function is called; subsequent calls skip over the definition, similar in principle to predicated execution. Once initialized, a static variable remains live until either the program terminates or execution reaches a place at which the function containing the static variable can no longer be called. We intend to properly address the handling of static variables in the future.

REFERENCES

[1] Beidas, R., and Zhu, J. Scalable interprocedural register allocation for high-level synthesis. *In Proc. of the 2005 Conference on Asia South Pacific Design Automation (ASP-DAC, '05)* (Shanghai, China, January 18-21, 2005) 511-516.

[2] Bouchez, F., Darte, A., Guillon, C., and Rastello, F. *Register Allocation and Spill Complexity Under SSA*, Technical Report 2005-33, ENS-Lyon, Lyon France, 2005.

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

[4] Briggs, P. *Register Allocation via Graph Coloring*. Ph.D. Thesis, Rice University, Houston, TX, USA, 1992.

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

[6] Chaitin, G. J. Register allocation and spilling via graph coloring. *In Proc. of the 1982 SIGPLAN Symp. on Compiler Construction*, (Boston, MA, USA, June 23-25, 1982), pp. 98-101.

[7] Chow, F. Minimizing register usage penalty at procedure calls. *In Proc. of the 1988 Int. Conf. Prog. Language Design and Implementation (PLDI '88)* (Atlanta, GA, USA, June 22-24, 1988) 85-94.

[8] Chudnovsky, M., Robertson, N., Seymour, P., and Thomas, R. The strong perfect graph theorem. *Technical Report* available online. Initial version, June 20, 2002; revised, July 19, 2005.

[9] Cooper, K. D., and Torczon, L. *Engineering a Compiler*. Morgan-Kaufmann, 2003.

[10] Gavril, F. 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*, June 1972, 180-187.

[11] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: a free commercially representative embedded benchmark suite. In *Proc. of the Workshop on Workload Characterization (WWC '01)*, (Austin, TX, USA, December, 2001), 3-14.

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

[13] Hack, S., Grund, D., and Goos, G. Register allocation for programs in SSA Form. *In Proc. of the 15th International Conf. on Compiler Construction (CC '06)* (Vienna, Austria, March 30-31, 2006) 247-262.

[14] Hashimoto, A. and Stevens, J. Wire routing by optimizing channel assignment within large apertures. *In Proc. of the 8th Workshop on Design Automation* (Atlantic City, NJ, USA, June 28-30, 1971) 155-169.

[15] Hind, M., Burke, M., Carini, P., and Choi, J-D. Interprocedural pointer alias analysis. *ACM Trans. Prog. Languages and Systems, vol. 21, no. 4*, July, 1999, 848-894.

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

[17] Kurdahi, F. J., and Parker, A. C. REAL: A program for Register Allocation. *In Proc. of the 24th ACM/IEE Conf. on Design Automation (DAC '87)* (Miami Beach, FL., USA, June 28 – July 1, 1987) 210-215.

[18] Lee, C., Potkonjak, M., and Mangione-Smith, W. H. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. *In Proc. of the 30th International Symposium on Microarchitecture (MICRO-30, '97)* (Research Triangle Park, NC, USA, December 1-3, 1997) 330-335.

[19] Matula, D. W., and Beck, L. L. Smallest last-ordering and clustering graph coloring algorithms. *J. ACM 30, 3*, July 1983, 417-427.

[20] Smith, M. D., and Holloway, G. *An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization*. Technical Report. Harvard University. 2002. Available online.

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

[22] Steenkiste, P. A., and Hennessy, J. L., A simple interprocedural register allocation algorithm and its effectiveness for LISP. *ACM Trans. Prog. Languages and Systems, vol. 11, no. 1*, January, 1989, 1-32.

[23] Stok, L. Transfer free register allocation in cyclic data flow graphs. In *Proc. of the European Conference on Design Automation (Euro-DAC '92)* (Brussels, Belgium, March 16-19, 1992) 181-185.

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

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

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

[27] Vemuri, R., Katkoori, S., Kaul, M., and Roy, J. An efficient register optimization algorithm for high-level synthesis from hierarchical behavioral specifications. *ACM Transactions on Design Automation of Electronoic Systesm, vol. 7, no. 1,* January 2002, 189-216.

[28] Zhang, S., and Dai, W. M. Linear time left edge algorithm. *Int. Conf. Chip Design Automation (ICCDA '00)* (Beijing, China, August 21-25, 2000).