Scalable Interprocedural Register Allocation for High Level Synthesis

Rami Beidas, Jianwen Zhu
Electrical and Computer Engineering
University of Toronto, ON M5S 3G4, Canada
{rbeidas, jzhu}@eecg.toronto.edu

Abstract— The success of classical high level synthesis has been limited by the complexity of the applications it can handle, typically not large enough to necessitate the departure from the industrial standard, register transfer level design methodology. Recent advances in micro-architecture model enabled the use of a stacked based controller, allowing complex algorithms with multiple procedures to be implemented directly in hardware. Nevertheless, design optimizations across procedure boundaries have not been fully explored. In this paper, we address the problem of interprocedural register allocation in the context of high level synthesis. In contrast to a recently proposed interprocedural register allocation algorithm, which processes an expensive, global, graph representation of the conflict relation of all values to achieve near optimality, we introduce a new method, called *color palette propagation* (CPP). The key idea behind our method, is to propagate the use of colors, whose number is significantly smaller than the size of the conflict relation, across different procedures. With a complexity comparable to intraprocedural register allocation, we show that our method can scale to very large C programs. For those benchmarks that can be handled by conventional global methods, our method produced nearly the same number of registers, while providing an average speedup factor of 90.

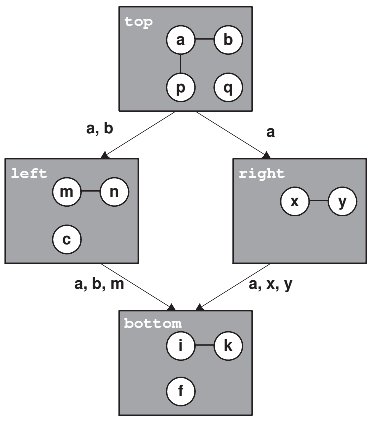
```
int top( int a, int b )
{
  int l, r, p, q;
  if( a > b )                  (1)
    l = left ( a, b );         (2)
  else
    r = right( a, a );         (3)
  p = a + 3;                   (4)
  q = p % a;                   (5)
  return q;                    (6)
}

int left( int m, int n )
{
  int c;
  if( m > 7)                   (7)
    c = n;                     (8)
  else
    c = bottom( m, m );        (9)
  return c;                    (10)
}

int right( int x, int y )
{
  x += bottom( x, y );         (11)
  return x;                    (12)
}

int bottom( int i, int k )
{
  int f = i + k;               (13)
  return ( f % 7 );            (14)
}
```

(a)



(b)

Fig. 1. Example of Interprocedural Register Allocation (a) Source Code (b) Call and Conflict Graphs

I. Introduction

High level synthesis (HLS) is the process of transforming a behavioral description of an algorithm to an optimized register transfer level (RTL) representation that implements the specified behavior [7, 13]. The HLS task is typically decomposed into subtasks of scheduling, register allocation, functional unit binding, and interconnect binding. The scheduling subtask determines the exact start control step for each operation, subject to original data and control dependencies, and possibly by resource constraints. For a scheduled design, register allocation determines a minimum grouping of variables of disjoint lifetimes to minimize register usage, and as a result, reduce target design area.

Research in HLS, however, has not been transformed into the industrial success envisioned by its pioneers. This is partly due to the fact that the complexity of the applications the classic HLS techniques can handle typically does not necessitate the departure from the industrial standard design methodology starting at the register transfer level. Advances in HLS micro-architecture [2, 3, 16, 20] have evolved to the point [9] where, with a stacked-based controller and a shared datapath, a complex behavior with multiple procedures can be directly synthesized into hardware. This new capability also offers new opportunities in design optimizations across the procedure boundaries, or *interprocedural optimizations*.

In this paper, we focus on the problem of interprocedural register allocation. While numerous efforts have attempted for the register allocation problem in the context of HLS, including [10, 15, 18, 22], they only focus on register allocation within the basic block or procedure boundary, or *intraprocedural register allocation*. With the traditional micro-architectures that synthesize separate hardware modules for different procedures [2, 3, 16, 20], intraprocedural register allocation can be directly employed. However, the implication here is that variables in different procedures, even though not alive at the same time, cannot be shared.

EXAMPLE 1. *Consider the C program in Figure 1. Here procedures* `left` *and* `right` *can safely share registers for their internal local variables. For example, when procedure* `top` *calls* `right`*, only variable* `a` *is live at the call site (indicated by the label of the corresponding call graph edge). As a result, local variables* `x` *and* `y` *of* `right` *can share registers with* `b`*,* `p`*, and* `q`*.* □

With a modern micro-architecture [9], where the datapath is shared between different procedures, intraprocedural register allocation can still be used directly, provided that a *calling convention* is used, such that when control is transfered across different procedures, the necessary register contents are saved, or *spilled* into memory. Such spilling cost can be eliminated if an interprocedural register allocator is used. In the context of HLS, Vemuri et al [21] proposed a solution that significantly outperforms the naive solution based on procedure inlining. However, their method cannot *scale* to large programs, since it requires the construction of conflict relation among all variables in the program, which could be prohibitively large.

In this paper, we propose a set of scalable, single-pass, interprocedural register allocation algorithms, based on a technique called *color palette propagation* (CPP). Our proposed algorithm achieves a significant, theoretic complexity reduction. The complexity of the algorithm in [21] is $\mathcal{O}(|V|^2)$, where V is the set of all values to be allocated. Due to the large value of $|V|$, this complexity is significantly larger than intraprocedural register allocation algorithm. In contrast, the complexity of our algorithm is comparable to intraprocedural register allocation, with only a modest overhead of $\mathcal{O}(|CG|\chi)$, where $|CG|$ is the number of call graph edges, and χ is the chromatic number, or the total number of register used. In practice, our algorithm can achieve hundred time speedup for the benchmarks for which [21] can complete.

The rest of the paper is organized as follows. Section II discusses related works in the compiler and CAD communities. Section III provides a formal definition and representation of the interprocedural optimization problem. Section IV provides a detailed description of the proposed allocation solution. Section V presents experimental results, followed by a conclusion in Section VI.

II. Background

The traditional register allocation problem in the compiler community is usually viewed as a graph coloring problem [14]. Equivalently, register allocation problem can be viewed as graph partitioning problem of *compatibility graphs*, where edges, instead, connect nodes whose objects' lifetimes do not overlap. Compatibility graphs are partitioned into cliques, each of which represent a single physical register. In HLS, a similar approach is used for register allocation, typically following the scheduling phase as part of the overall process of resource sharing and binding [13]. The main difference is the availability, in theory, of unlimited allocation resources, rather than K registers, with the primary optimization goal of reducing the area of the target design. An excellent review of past efforts can be found in [21].

Most techniques reported limit their optimization scope to a single procedure, or are *intraprocedural*. To adapt to multiprocedural programs in software compilers, the simplest approach relies on the spilling of registers that might be used by both a caller and a callee. An alternative approach, referred to as *interprocedural* register allocation, would consider the requirements of each procedure and relations among procedures of a target application, to minimize spilling and, as a result, execution cost. Static interprocedural optimizations, such as the allocators developed by Chow [5] and by Steenkiste and Hennessy [19], usually rely on a bottom-up traversal of the call graph, utilizing unused registers in callees when processing callers, sharing registers between procedures that cannot be active at the same time. Chow's allocator spills variables based on a priority model, while Steenkiste's uses spilling when running out of registers in the bottom up traversal, where spilling would hopefully occur for less frequent calls at higher levels in the call graph. Apart from the need to satisfy different requirements for software compiler, these bottom-up approaches can be considered as special cases of our solution. With the generalized application of our proposed CPP technique, the procedures can be colored in an arbitrary order determined by program profiling, thereby achieving the spilling cost reduction not possible by the bottom-up method [1].

In HLS synthesis, interprocedural register allocation could be even more crucial. As no complex applications would avoid the use of procedure calls, the inability to efficiently share registers among specification procedures would result in excessive waste of circuit resources. A carrier-based interprocedural solution was recently proposed by Vemuri et al [21]. The proposed solution traverses the call graph in a bottom-up fashion, using intraprocedural compatibility graph to build a global compatibility graph, whose partitioning would return an optimized feasible allocation solution.

A main shortcoming of most interprocedural solutions in most compiler is accuracy [19]. At any call site within a procedure, all local variables are assumed to be live, disallowing the possibility of sharing between a caller and a callee, which could easily result in far-from-optimal solution. Although [21] overcomes this weakness, the fact that a global, probably dense, compatibility graph needs to be built and partitioned poses a limiting factor for the scalability of the proposed solution (an average speedup of only 7.0 was reported when compared to a method based on inline expansion).

III. Problem Definition

The input of our problem is a sequential program consisting of a set of procedures, each represented in the scheduled, static single assignment form (SSA) [6]. In SSA, all register allocation targets, including local and temporal variables, are assigned, or *defined* exactly once. Therefore, operations (or instructions) in the program have a one-to-one mapping to the set of values they define. In the text following, we use the term value exclusively.

Starting with a scheduled SSA, liveness analysis can be performed to identify live ranges for each SSA value. Liveness analysis is a traditional lattice-based data-flow analysis that identifies for each variable the set of program points at which the current value of a variable may be used before redefinition [14]. The application of liveness analysis on SSA intermediate representation associates a separate live range with each value, as opposed to traditional intermediate representation where different values of a variable may be associated with the same live range. Note the use scheduled control steps as the basic components of live ranges, while traditional compilers use program points instead.

Definition 1 gives a formal model of the problem input.

Definition 1. *The input to an interprocedural register allocation problem is a tuple $\langle P, V, CG, LR, O\rangle$, where:*

- *P is the set of procedures;*
- *V is the set of values in P;*
- *$CG : P \times V \times P$ is the call graph, or the set of call sites $\{\langle p, v, q\rangle\}$, where p is the caller, v is the return value, and q is the callee;*
- *$LR : V \mapsto 2^{\mathbb{N}}$ defines the live range of each value as the set of control steps during which that value is alive;*
- *$O : V \mapsto P$ maps each value to the owner procedure where it is defined.* □

Within our framework, we can derive two types of conflicts; *local* or intraprocedural conflicts, and *global* or interprocedural conflicts. The traditional intraprocedural conflicts are defined in Definition 2.

Definition 2. *The set of local conflicts $LC : V \times V$ identifies a relation such that $\langle u, v\rangle \in LC \Leftrightarrow (O(u) = O(v)) \wedge (LR(u) \cap LR(v) \neq \oslash)$.* □

In other words, two variables of a single procedure conflict if their live ranges, defined by the schedule, overlap. As mentioned before, such conflicts are usually captured by conflict graphs, which become the focus of intraprocedural register allocation, where allocation is simply the solution of a graph coloring problem [4, 8, 10, 12, 18, 21]. Different coloring algorithms define different *elimination orders*, the order in which nodes are colored. The complexity of an algorithm and the quality of a returned elimination order depends heavily on interconnection properties of the conflict graph as well as the accuracy of the coloring solution.

Interprocedural conflicts are defined in two steps; *immediate global conflicts* (IGC), and *global conflicts* (GC), formally defined as follows.

DEFINITION 3. *The set of immediate global conflicts* $IGC : V \times V$ *identifies a relation such that* $\langle u, v\rangle \in IGC \Leftrightarrow \exists \langle p, w, q\rangle \in CG : p = O(u) \wedge p = O(v) \wedge \langle u, w\rangle \in LC$. $\Box$

DEFINITION 4. *The set of global conflicts* $GC : V \times V$ *identifies a relation such that* $\langle u, v\rangle \in GC \Leftrightarrow (\langle u, v\rangle \in IGC) \vee (\exists w \in V : \langle u, w\rangle \in IGC \wedge \langle w, v\rangle \in GC)$. $\Box$

IGCs exist between a caller value and callee values if the caller value is live at a call site to the callee. On the other hand, GCs cover the transitive property of interprocedural conflicts such that a caller value live at a call site conflicts with all transitive callee values. Note that this transitive property is not applicable to local conflicts. This interprocedural conflict information is partially built during the process of liveness analysis of each procedure. As liveness analysis proceeds, a record is kept of the set of values live at the current control step. If the current control step contains a procedure call instruction, the current set of live values are saved for further processing.

Now, we can formally define a legal interprocedural register allocation as follows:

DEFINITION 5. *Interprocedural register allocation is a function* $RA : V \mapsto \mathbb{N}$ *such that* $\forall u, v \in V : \langle u, v\rangle \in LC \vee \langle u, v\rangle \in GC \Longrightarrow RA(u) \neq RA(v)$. $\Box$

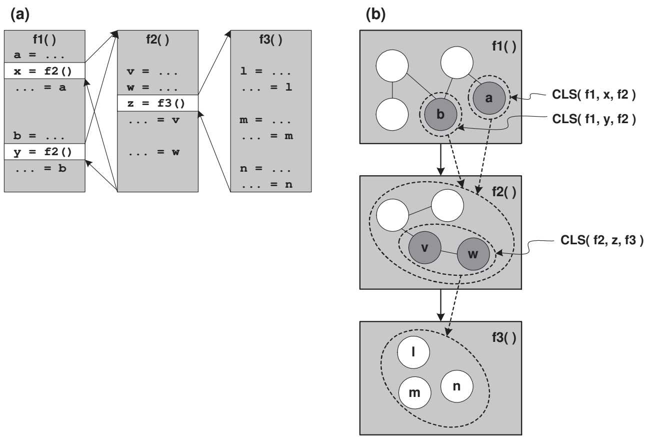


Fig. 2. An Example of Interprocedural Conflict Graphs (a) a simple program (b) program's conflict graph

Definitions listed above are best illustrated through an example.

EXAMPLE 2. *Consider the representation of the simple program shown in Figure 2 (a). Live ranges of values* `v` *and* `w` *within* `f2` *clearly overlap, therefore* $\langle v, w\rangle \in LC$. *On the other hand, value* `a` *of* `f1`, *which is live at a call site of* `f2`, *conflicts with every value in* `f2` *including* `v` *and* `w`, *therefore* $\langle a, v\rangle, \langle a, w\rangle \in IGC \subseteq GC$. *Similarly,* $\langle v, l\rangle, \langle v, m\rangle, \langle v, n\rangle \in IGC$. *Using the transitive property of interprocedural conflicts, we also conclude that* $\langle a, l\rangle, \langle a, m\rangle, \langle a, n\rangle \in GC$. *Remaining conflicts can be derived in the same manner. Interprocedural register allocation is a value-to-register mapping that would respect all the conflicts defined in* LC *and* GC.

For convenience, we also define the concept of *call live sets* (CLS), which identifies the set of live values at each call site.

DEFINITION 6. *A call live set* $CLS : CG \mapsto V$ *is a set of values such that* $\forall u \in CLS(p, v, q), w : O(w) = q$, *then* $\langle u, w\rangle \in GC$. $\Box$

For convenience, sometimes we omit the u in the $CLS(p, u, q)$ in our illustrations. When the context is clear, and use $CLS(p, q)$ instead.

EXAMPLE 3. *In the simple program of Figure 2,* $CLS(f1, x, f2) = \{a\}$, $CLS(f1, y, f2) = \{b\}$, *and* $CLS(f2, z, f3) = \{v, w\}$.

In the rest of the paper, we illustrate the local and global conflicts using an interprocedural conflict graph represented as follows. In this graph, each value $v \in V$ is associated with a node, each local conflict $\langle u, v\rangle \in LC$ is represented with an edge, and interprocedural conflicts are associated with special edges connecting $CLS(p, w, q)$ with the set of all values of the callee q identifying an immediate global conflict between each value $u \in CLS(p, w, q)$ and each value $v \in V : O(v) = q$. In Figure 2 (b), each node represents a value, and local conflicts are represented by solid edges, and immediate global conflicts are represented by dotted edges connecting value sets. Non-immediate global conflict edges, such as the leftmost dotted edge, are omitted for simplify the figure.

IV. COLOR PALETTE PROPAGATION-BASED INTERPROCEDURAL REGISTER ALLOCATION

A key idea behind our interprocedural register allocation algorithm is to avoid the construction of the massive global conflict graph, while leveraging the mature intraprocedural register allocation algorithm as much as possible. We achieve this goal by using a technique called *Color Palette Propagation* (CPP). Instead of directly propagating across procedures the coloring constraints in terms of conflict relation , CPP propagates constraints in terms of the available colors, called *palette*, which is much more compact. This strategy is responsible for the scalability of our algorithm.

In the sequel, we start by modifying the traditional intraprocedural register allocation algorithm so that it can accommodate the palette constraints. We will then present three interprocedural register allocation algorithms, each following a different traversal order of the call graph, to derive the palette constraints and drive the intraprocedural register allocation.

A. Constrained Intraprocedural Register Allocation

The intraprocedural register allocation algorithm is no different from the traditional coloring algorithm, except that it uses $pallette$, the set of available colors for each value in the program, as an additional constraint. As shown in Algorithm 1, in takes as inputs the procedure p to color, its local conflict relation LC, and a coloring order σ. It updates relevant register assignment in $color$, and also records the set of used colors in $used$. According to the order given by σ, it colors one value at a time. To color a value v, it first removes from its palette all the colors that have been used by its colored neighbors. It then picks an available color from the palette, and updates $color$ and $used$ accordingly.

B. Top-Down Interprocedural Register Allocation

One strategy is to color the procedures in the program in a top-down fashion by traversing the call graph is a topological order. We assume that this order is given by Σ as the input to Algorithm 2. The

ALGORITHM 1. *Intraprocedural register allocator.*

```
var palette : V ↦ 2^N;
var used : P ↦ 2^N;
var color : V ↦ ℕ;

intraColor = func( p ∈ P, LC : V × V, σ : [ ]^V ) {
  foreach( v ∈ σ ) {
    foreach( w ∈ V : ⟨v, w⟩ ∈ LC ∧ w <_σ v )
      palette(v) = palette(v) \ {color(w)};
    color(v) = palette(v)[0];
    used(p) = used(p) ∪ {color(v)};
    }
  }
```

ALGORITHM 2. *Top-down interprocedural register allocation.*

```
var top : P ↦ 2^N;

interColorTopDown = func( Σ : [ ]^P ) {
  var LC : V × V;
  var σ : [ ]^V;
  var CLS : CG ↦ 2^V;

  foreach( p ∈ P )
    top(p) = ⊤;

  ⟨LC, CLS⟩ = livenessAnalysis();
  foreach( p ∈ Σ ) { // Σ is in topological order of CG
    σ = buildElimOrder( p, LC );

    foreach( v ∈ V : O(v) = p )
      palette(v) = top(p);
    intraColor( LC, σ, palette );

    foreach( ⟨p, v, callee⟩ ∈ CG )
      propagateTopDown( p, callee, CLS(p, v, callee) );
    }
  }
propagateTopDown = func( p : V, callee : V, live : 2^V ) {
  top(callee) = top(callee) ∩ top(p);
  foreach( v ∈ live )
    top(callee) = top(callee) \ {color(v)};
  }
```

algorithm starts by first performing standard liveness analysis to obtain the local conflict LC, and the call live set CLS. For each procedure p, a coloring order σ is derived using an algorithm such as Chaitin's [4]. We associate each procedure p with a palette $top(p)$, representing the palette constraint propagated from its ancestors. This palette is initially full. This palette constraint for the procedure is translated to the constraint for each value contained in procedure p, before the intraprocedural coloring algorithm is invoked. Once the values of the current procedure are colored, the palette constraints are propagated to its *immediate* callees. At each call site $\langle p, v, callee\rangle$, the algorithm calls $propagateTopDown$, which first intersects the palette of $callee$ from that of the caller p, as every color unavailable to a caller is also unavailable to a callee due to the transitive nature of GCs. The colors used by the corresponding call live set $CLS(p, v, callee)$ is then removed from the palette of $callee$, to cover IGCs. The topological order of procedure precessing, ensures that all callers of a procedure p are processed before p, and p would have an appropriate palette that takes into account all the global conflicts defined in Definition 4.

Note that at any point in time, only the local conflict graph of a single procedure is processed, a crucial property that would have a major effect on the scalability and efficiency of our proposed solution.

EXAMPLE 4. *A detailed example of top-down CPP is shown in Figure 3. Given that* `f1` *is the root procedure, it has a full color palette,* $top(f1) = \top$, *and the conflict graph nodes are colored with no restrictions. When the palette is propagated to* `f2`, *color 1 is marked unavailable, as CLS(* `f1, f2` *) =* $\{a\}$ *is colored using color 1. Similarly, color 3 is marked unavailable on the palette propagated to* `f3`. *In case of* `f4`, *its palette would be that of* `f2`, *its only parent, with colors 2 and 4 marked unavailable as CLS(* `f2, f4` *) =* $\{c, d\}$ *is colored using those colors. Finally, the palette of* `f5` *would be the intersection of its parents palettes, as colors unavailable to parents are unavailable to children, with color 2 and 3 marked unavailable, as CLS(* `f2, f5` *) =* $\{d, e\}$ *is colored using colors 2 and 3. Color 1 is unavailable for two reasons; CLS(* `f3, f5` *) =* $\{g\}$ *is colored using color 1, also color 1 is unavailable to a parent,* `f2` *in this case.*

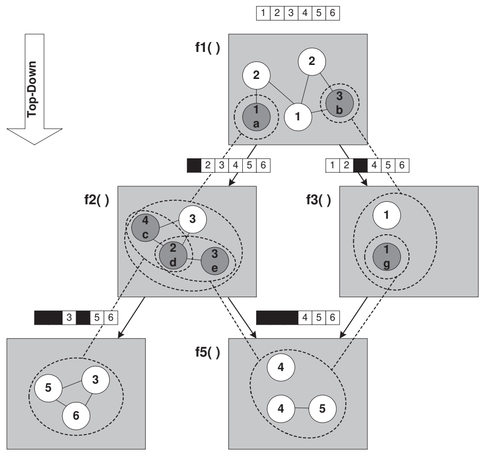


Fig. 3. Top-Down Color Palette Propagation

C. Bottom-Up Interprocedural Register Allocation

Another strategy is to color the procedures in the program in a bottom-up fashion by traversing the call graph in a reversed topological order. This order is available in Algorithm 3 as Σ.

As in the case of top-down algorithm, the bottom-up algorithm starts by liveness analysis, during which the set of local conflicts, LC, is generated and the set of call live sets, CLS, is identified. Thanks to the traversal order, when a procedure p is processed, $palette$ already contains all the necessary palette constraints imposed by all its callees. The intraprocedural coloring algorithm is invoked after a coloring order is found. Once the values of the current procedure p are colored, the interprocedural conflict relations are propagated to its callers, again through CPP: for each call site $\langle caller, v, p\rangle$, $propagateBottomUp$ is invoked such that the $used(p)$, or the colors used by p and its descendents, is removed from $used(caller)$, and in addition, is removed from the palettes of all values in the call live set $CLS(caller, v, p)$.

Note again that here we only need to process a local conflict graph. Also note in particular that under the bottom-up algorithm, the palette defined by $used(p)$ of any procedure p is guaranteed to be a continuous sequence $[0, M]$. Since the set can be characterized by a single number M, the implementation can be *simplified* to avoid the set operation shown in Algorithm 3.

A major difference between the top-down and the bottom-up propagation is in the way they handle multiple incoming (caller) and outgoing (callee) palettes, respectively. In case of top-down propagation, the palette constraints come from the colors used in the call live sets

ALGORITHM 3. *Bottom-up interprocedural register allocation.*

```
interColorBottomUp = func( Σ : [ ]^P ) {
  var LC : V × V;
  var σ : [ ]^V;
  var CLS : CG ↦ 2^V;

  ⟨LC, CLS⟩ = livenessAnalysis();
  foreach( p ∈ Σ ) { // Σ is in inverse topological order of CG
    σ = buildElimOrder( p, LC );

    colorIntra( p, LC, σ );
    foreach( ⟨caller, v, p⟩ ∈ CG )
      propagateBottomUp( p, caller, CLS(caller, v, p) );
    }
  }
propagateBottomUp = func( p, caller : V, live : 2^V ) {
  used(caller) = used(caller) ∪ used(p);
  foreach( v ∈ live )
    palette(v) = pallette(v) \ used(p);
  }
```

of all ancestors, and are applied to all values in the procedure. On the other hand, in case of bottom-up propagation, the palette constraints come from the colors used in all descendants, and are applied to values in the corresponding call live sets.

EXAMPLE 5. *A detailed example of bottom-up CPP is shown in Figure 4. Given that* `f4` *and* `f5` *are leaf procedures, they have full color palettes, and their conflict graph nodes are colored with no restrictions. At this stage,* $used(f4) = \{1, 2, 3\}$ *and* $used(f4) = \{1, 2\}$. *When processing* `f3`, *CLS(* `f3`, `f5` *)* $= \{g\}$ *has a color palette with colors 1 and 2 marked unavailable, which are the colors used in* `f5`. *Note that in bottom-up propagation values of different, possibly overlapping, CLS's could have different color palettes. For example, CLS(* `f2`, `f4` *)* $= \{c, d\}$ *has a color palette with colors 1, 2, and 3 marked unavailable due to their use in* `f4`, *while CLS(* `f2`, `f5` *)* $= \{d, e\}$ *has a color palette with colors 1 and 2 marked unavailable. The same scenario applies to* `f1`, *where CLS(* `f1`, `f2` *)* $= \{a\}$ *has a color palette with colors 1 to 5 marked unavailable, while CLS(* `f1`, `f3` *)* $= \{b\}$ *has a color palette of with colors 1 to 3 absent. Finally, remaining nodes of any procedure that do not belong to any CLS can be colored freely as long as local conflicts are respected.*

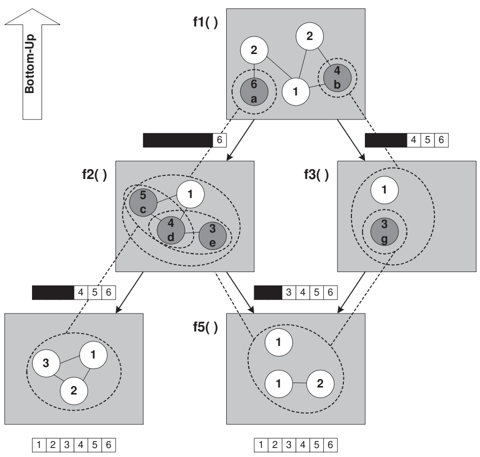


Fig. 4. Bottom-Up Color Palette Propagation

D. Algorithm Complexity

In this section, we derive the complexity of the overhead, with respect to the intraprocedural register allocation algorithm, that our interprocedural algorithms introduce. Note that the $livenessAnalysis$, $buildElimOrder$ in our algorithms are exactly the same as the intraprocedural register algorithm. Our $intraColor$ is slight different due to the use of palette constraints, but this difference does not change the complexity. Our overhead therefore comes only from palette propagation.

$propagateTopDown$ has a time complexity of $\mathcal{O}(\chi)$, where χ is the chromatic number. This is first due to the fact that the set intersection in line 36 is linear to the set size, which is χ. Since the cardinality of any call live set is bounded by χ, and the member inclusion operation in line 39, is of $\mathcal{O}(1)$, the loop in 38-39 is also of $\mathcal{O}(\chi)$. Therefore, the overhead for our top-down interprocedural register allocation algorithm is of $\mathcal{O}(|CG|\chi)$.

Based on the same argument, the simplified implementation of $propagateBottomUp$, is of $\mathcal{O}(\chi)$. Therefore, the overhead of our interprocedural register allocation algorithm, where the simplified implementation is valid, is guaranteed to be of $\mathcal{O}(|CG|\chi)$.

It is important to note that the chromatic number χ is not a large number in practice. It is reasonable to assume that χ is a constant, in which case the complexities of our algorithms becomes $\mathcal{O}(|CG|)$ for top-down and bottom-up. In other words, the interprocedural overhead is only *linear* to the number of call sites in the program. In practice, this overhead is only a small faction of the time spent on intraprocedural register allocation. This is in sharp contrast with the previous approach based on global compatibility graph, which effectively is of $\mathcal{O}(|V|^2)$, due to the dense nature of the global conflict graph. This performance gap will be affirmed by the empirical results of Section V.

V. EXPERIMENTAL RESULTS

In this section we present experimental results comparing runtime and quality of results of our approach compared to the global solution using benchmarks from the integer suite in SPEC2000 [17] and MediaBench [11] to illustrate the scalability and efficiency of the proposed solution.

We have implemented the proposed solution in C using appropriate data structures as part of a complete HLS suite. The experiments were performed on a Sun Blade 150 workstation with 550 MHz CPU and 128MB RAM, running on Solaris 8 Operating System.

Throughout our experiments, we assume a sequential execution of target benchmarks for allocation results to serve as a reference for alternative allocation solutions.

Table I, Figure 5, and Figure 6 show experimental results on SPEC2000 and MediaBench benchmarks. For each benchmark we list program statistics, namely number of lines of C code and number of procedures as measures of the complexity of the analyzed programs. We report results of our CPP-based top-down and bottom-up implementations in comparison with a global interprocedural optimization, to show the speedup in execution times and quality of results in terms of the number of obtained registers. We also report the overhead time over mere intraprocedural register allocation. All reported execution times in Table I are in seconds, and NA indicates the inability to report results due to scalability limitation of the global solution.

From the reported results we can draw the following observations:

- CPP-based solutions are scalable, capable of processing benchmarks as large as vortex of 52,637 lines of code.
- The advantage of global optimization on tested benchmarks did not exceed an average of 7.9% of the total number obtained registers.

Benchmark				Global	Top-Down			Bottom-Up		
Package	Appl	LOC	#Procs	Total Time	Overhead Time	Total Time	Speedup	Overhead Time	Total Time	Speedup
SPEC2000	bzip2	4665	63	108.01	0.18	16.62	6.50	0.02	16.46	6.50
	crafty	19478	104	NA	11.45	573.89	NA	0.23	562.67	NA
	parser	11391	297	NA	1.97	54.73	NA	0.07	52.83	NA
	twolf	19756	167	NA	4.88	939.19	NA	0.16	934.47	NA
	vortex	52637	600	NA	51.29	472.41	NA	0.33	421.45	NA
	vpr	16984	281	NA	2.68	45.83	NA	0.07	43.22	NA
MediaBench	epic	3339	26	4.24	0.02	1.16	3.66	0.01	1.15	3.69
	mpeg2d	8680	113	1275.30	0.15	6.49	196.50	0.04	6.38	199.89
	mpeg2e	6801	95	3308.32	0.08	21.77	151.97	0.05	21.74	152.18
	pgp	28065	256	NA	1.34	33.14	NA	0.05	31.85	NA
	rasta	6951	65	108.94	0.38	32.65	3.34	0.02	32.29	3.37

TABLE I
INTERPROCEDURAL REGISTER ALLOCATION RUNTIME FOR SPEC2000 AND MEDIABENCH BENCHMARKS

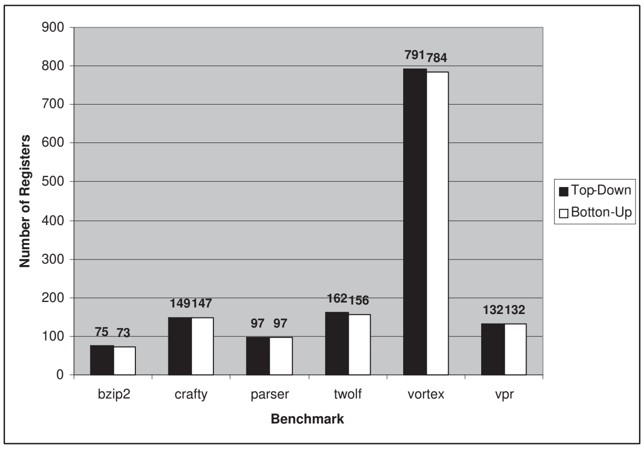


Fig. 5. Interprocedural Register Allocation Results for SPEC2000 Benchmarks

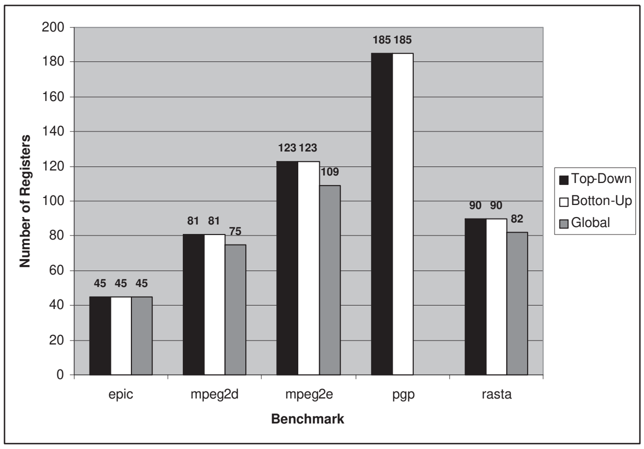


Fig. 6. Interprocedural Register Allocation Results for MediaBench Benchmarks

- In addition to the near optimal results, with respect to global optimization, our bottom-up CPP-based solutions provided and average speedup of 90 on MediaBench benchmarks when compared to the global implementation, not to mention the benchmarks that could not be handled by the global solution. Note that the speedup is in fact higher if we ignore the contribution of liveness analysis to both results.
- Top-down and bottom-up implementations are almost equivalent in runtime and quality of results, with a slight edge of the bottom-up implementation mainly due to the absent of fragmented color palettes.
- The overhead of interprocedural register allocation when compared to intraprocedural is in fact negligible ($< 1\%$ for the bottom-up solution).

VI. CONCLUSION

In this paper we argued the need for interprocedural register allocation in the context of HLS, pointed out the shortcomings of traditional solutions, and proposed a new interprocedural register allocation algorithm based on a new conflict propagation method. Based on our study, we conclude that the proposed solution is effective, as it produces near-optimal results, and is scalable, as it can complete in seconds even for benchmarks that are much larger than those usually considered as candidates for HLS.

VII. REFERENCES

[1] R. Beidas and J. Zhu. Scalable register allocation for high level synthesis. Technical Report TR-01-11-04, Electrical and Computer Engineering, University of Toronto, November 2004.

[2] R. Camposano, L. Saunders, and R. Tabet. VHDL as input for high level synthesis. *IEEE Design and Test of Computers*, pages 43–49, March 1991.

[3] R. Camposano and J. van Eijndhoven. Partitioning a design in structural synthesis. 1987.

[4] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings SIGPLAN Symposium on Compiler Construction*, June 1982.

[5] F. C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1988.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[7] D. D. Gajski, N. D. Dutt, A. Wu, and S. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, January 1992.

[8] K. Gopinath. Register allocation. *The Compiler Design Handbook*, September 2002.

[9] K. Jasrotia and J. Zhu. Stacked FSMD: A power efficient micro-architecture for high level synthesis. In *International Symposium on Quality Electronics Design*, March 2004.

[10] F. Kurdahi and A. Parker. REAL: A program for register allocation. In *Proceeding of the 24th Design Automation Conference*, June 1987.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, November 1997.

[12] G.-Y. Lueh. Issues in register allocation by graph coloring. Technical Report CMU-CS-96-171, School of Computer Science, Carnegie Mellon University, November 1996.

[13] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, January 1994.

[14] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, July 1997.

[15] P. G. Paulin and J. P. K. E. F. Girczyc. HAL: A multi-paradigm approach to automatic data path synthesis. In *Proceeding of the 23rd Design Automation Conference*, July 1986.

[16] L. Ramachandran, S. Narayan, F. Vahid, and D. Gajski. Synthesis of functions and procedures in behavioral VHDL. In *Proceedings of the European Design Automation Conference*, 1993.

[17] SPEC2000. *Standard Performance Evaluation Corporation*. `http://www.specbench.org/cpu2000/`.

[18] D. L. Springer and D. E. Thomas. Exploiting the special structure of conflict and compatibility graphs in high-level synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, November 1990.

[19] P. A. Steenkiste and J. L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for lisp. *ACM Transactions of Programming Languages and Systems*, 11(1), January 1989.

[20] F. Vahid. Procedure exlining: A new system-level specification transformation. In *European Design Automation Conference*, pages 508–513, September 1995.

[21] R. Vemuri, S. Katkoori, M. Kaul, and J. Roy. An efficient register optimization algorithm for high-level synthesis from hierarchical behavioral specifications. *ACM Transactions on Design Automation of Electronic Systems*, 7(1), January 2002.

[22] N.-S. Woo. A global, dynamic register allocation and binding for a data path synthesis system. In *Proceeding of the 27th Design Automation Conference*, June 1990.