Optimal Register Sharing for High-Level Synthesis of SSA Form Programs

Philip Brisk, Foad Dabiri, *Student Member, IEEE*, Roozbeh Jafari, *Student Member, IEEE*, and Majid Sarrafzadeh, *Fellow, IEEE*

***Abstract*—Register sharing for high-level synthesis of programs represented in static single assignment (SSA) form is proven to have a polynomial-time solution. Register sharing is modeled as a graph-coloring problem. Although graph coloring is NP-Complete in the general case, an interference graph constructed for a program in SSA form probably belongs to the class of chordal graphs that have an optimal $O(|V| + |E|)$ time algorithm. Chordal graph coloring reduces the number of registers allocated to the program by as much as 86% and 64.93% on average compared to linear scan register allocation.**

***Index Terms*—Compilers (silicon), high-level synthesis.**

I. Introduction

REGISTER sharing during high-level synthesis is proven to have an optimal solution if the intermediate representation is a scheduled control flow graph (CFG) in static single assignment (SSA) form. Register sharing allows program variables with nonoverlapping lifetimes to reside in the same register. Without register sharing, each variable in the intermediate representation is stored in a separate register. An optimal solution to the register sharing problem minimizes the number of registers in the resulting data path, yielding a more compact design with increased register utilization.

Register sharing has historically been modeled as a graph-coloring problem. For a program represented as a CFG in SSA form, the graph to be colored is proven to have an optimal solution with an $O(|V| + |E|)$ time complexity. Results for 12 large procedures in SSA form show that optimal register sharing reduces the number of registers allocated to the design by 64.93% (42.83 registers) on average compared to linear scan register allocation [1]–[3]. The per-benchmark percentage reduction ranged from 8.33% to 86.36% (1 to 87 registers).

II. High-Level Synthesis and Register Sharing Overview

High-level synthesis is the transformation of an intermediate representation of a computation into a structural description of a data path that implements it. A high-level synthesis system contains many stages, including, but not limited to, resource allocation (computational, storage, and interconnect), scheduling, resource selection, resource sharing, binding operations to specific resource instances, and clock selection. These problems are interdependent, and different formulations exist for each depending on the order in which they are solved. Many of these problems are also NP-Complete.

A procedure body is represented as a CFG, a directed graph where nodes represent basic blocks—maximal sequences of a straight line code with no branches or branch targets interleaved—and edges represent transfers of control from the end of one block to the beginning of another. Each basic block is represented by a directed acyclic graph (DAG) called a data flow graph (DFG). A typical synthesis flow will begin by allocating computational resources (adders, multipliers, etc.) to the design and scheduling each operation on each resource. The scheduled DFG can then be represented by a list of sets of operations; the ith set contains all of the DFG operations that were scheduled at the ith time step.

The problem of register sharing is typically formulated and solved following resource allocation and scheduling. Let V be the set of variables in the program. An interference graph $G = (V, E)$ is defined, where $(u, v) \in E$ indicates that variables u and v interfere, i.e., their lifetimes overlap and thus require separate storage resources.

An independent set is a subset $V' \subseteq V$ such that there are no edges between any pair of vertices in V'. A k-coloring of G is a partition of V into k nonoverlapping independent sets (color classes): $\{C_1, C_2, \ldots, C_k\}$. A k-coloring can also be represented by a function $f : V \rightarrow \{1, 2, \ldots, k\}$, defined such that $f(v) = j$ is equivalent to the statement $v \in C_j$. k-coloring ensures that for every edge $(u, v) \in E$, $f(u) \neq f(v)$. The goal of register sharing is to color G with the fewest possible colors. Each color class represents a set of variables with nonoverlapping lifetimes that share the same register.

In early synthesis systems, inputs were limited to DFGs. For a scheduled DFG, the interference graph for register sharing belongs to the class of interval graphs, which can be colored optimally in $O(|V|log|V|)$ time using the Left Edge Algorithm [2], [3]; since all interval graphs are chordal, an $O(|V| + |E|)$-time coloring algorithm [4] could also be used.

Springer and Thomas [5] showed that chordal interference graphs arise during synthesis if certain conditions are imposed on variable lifetimes with respect to conditional branches, merge points, and module calls. Their work predated the adoption of the SSA form for synthesis.

When cycles exist in the intermediate representation, either due to loop constructs or recursive module calls, then the

Manuscript received July 15, 2005; revised October 14, 2005. This paper was recommended by Guest Editor R. I. Bahar.

The authors are with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: philip@cs.ucla.edu; dabiri@cs.ucla.edu; rjafari@cs.ucla.edu; majid@cs.ucla.edu).

Digital Object Identifier 10.1109/TCAD.2006.870409

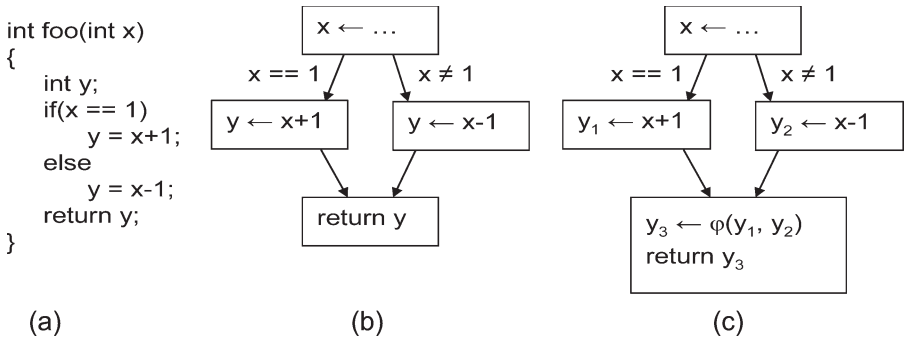

Fig. 1. (a) Small procedure represented as (b) CFG and shown in (c) SSA form.

interference graph belongs to the class of circular arc graphs. Circular arc coloring is NP-Complete. Stok [6] has suggested that this circular arc coloring should be solved via transformation to a multicommodity flow problem.

III. SSA Form

Any operation of the form $v \leftarrow \ldots$ defines variable v; any operation of the form $\ldots \leftarrow v$ uses v. For simplicity, assume that each instruction defines at most one value. A CFG is in SSA form if each variable is: 1) defined once and 2) each use corresponds to one definition. Prior to register allocation, compilers and synthesis tools assume an infinite supply of registers. With limitless registers, criterion 1) is trivially satisfied. Satisfying the criterion 2) entails the insertion of φ functions where different control paths in the program merge.

Fig. 1 shows a procedure *foo* (a) represented as a CFG (b) and in SSA form (c). In the CFG, variable y is defined on both sides of a condition and used later at the join point. In SSA form, the definitions of y have been renamed to y_1 and y_2. Depending on the whether condition is true or false, the use of y at the join point will be either a use of y_1 or y_2, not both. A φ function $y_3 \leftarrow \varphi(y_1, y_2)$ merges the definitions of y_1 or y_2 into a new variable y_3 to represent this conditional use. y_3 explicitly represents the fact that different definitions of y occur on paths that merge together in the original CFG. Without the φ function, the use of y in the return statement has two definitions, one on each side of the condition.

It is possible to directly synthesize a CFG in SSA form by realizing φ functions in hardware as multiplexers [7]. In Fig. 1, y_3 will receive its value from y_1 or y_2, depending on which control path entering the CFG node containing the φ function is taken at runtime.

It is not possible, however, to compile a program in SSA form without first translating it out of SSA form. This is necessary because the vast majority of modern architectures offer no assembly instruction that meets the abstract specification of a φ function. Conditional move (cmove) and conditional select (csel) instructions offer similar functionality, but can only select between a fixed number of inputs. φ functions must select between k variables, where k is the number of predecessors of the basic block containing the φ function. Since k can grow arbitrarily large, encoding and decoding φ functions in hardware is likely to be unwieldy.

IV. Strict Programs

A strict program [8] ensures that every variable is assigned a value before the variable is used in a computation along every possible path of program execution. Some languages impose strictness by definition. In Java, for example, all variables that are not explicitly initialized by the programmer are implicitly initialized to 0 by the compiler. Other languages, for example C/C++, do not impose strictness as a requirement. Most real-world programs are strict, regardless of language; however, nonstrict programs do exist, and compilers and synthesis tools must be able to handle them.

A regular program [8] is a strict program in SSA form. A proof is given in the Appendix that an interference graph constructed for a regular program belongs to the class of chordal graphs. This is a notable result because graph coloring can be solved in $O(|V| + |E|)$ time for chordal graphs.

The theoretical result proven in this paper requires a regular program to ensure correctness. Nonstrict programs explicitly violate the SSA form since some variables that are not defined may exist. Compilers such as the GNU C Compiler (GCC) insert implicit definitions for all variables that are used but not defined. Implicit definitions can also be used to correct the case where a variable is defined on some (but not all) paths that converge at a φ function. This imposes strictness on the intermediate representation, even if the input program is not strict. Throughout the duration of this paper, it is assumed that programs in SSA form are regular.

V. Liveness Analysis for Programs in SSA Form

According to Cooper and Torczon [9, p. 630], "variable v is live at point p if it has been defined along a path from the procedure's entry to p and there exists a path from p to a use of v along which v is not redefined." Liveness analysis is the process of computing the set of variables that are live at each point in the CFG. To build an interference graph for a CFG in SSA form, a minor modification to standard liveness analysis is required. We have implemented and modified the liveness analyzer described by Cooper and Torczon [9, pp. 437–447].

If n is a basic block in a CFG, then LIVEOUT(n) is the set of all such variables that are live upon exiting n. Intuitively, LIVEOUT(n) contains those variables that are defined either in n or some other node n' from which n is reachable, and are used in some CFG node n'' reachable from n. Liveness analysis

computes LIVEOUT(n) for every basic block in the program. To compute LIVEOUT(n), two additional sets of variables are required: UEVAR(n) and VARKILL(n).

UEVAR(n) is defined to be the set of upward-exposed variables in n. UEVAR(n) contains all the variables that are used in n but are not defined in n; some block that precedes n during program execution must define each variable in UEVAR(n). VARKILL(n) is the set of all variables that are defined in n. Both UEVAR(n) and VARKILL(n) can be constructed by a linear traversal of the operations in n.

Let succ(n) be the set of successor CFG nodes of n. In other words, there must be some (conditional or unconditional) control transfer from n to each node in succ(n). LIVEOUT(n) is defined recursively as

$$\text{LIVEOUT}(n) = \bigcup_{m \in \text{succ}(n)} \text{UEVAR}(m) \cup \left(\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}\right). \quad (1)$$

During liveness analysis, (1) is repeatedly solved for each basic block in the program until stability is reached.

For programs in SSA form, let preds(m) be the set of predecessors of CFG node m. Then, for each CFG node $n \in$ preds(m), UEVAR(m) is replaced with UEVAR(m, n): the set of upward-exposed variables from m to n. This allows m to expose different φ function parameters to predecessors on different incoming control paths, i.e., mutual exclusion. There may also be variables that are originally in the UEVAR(m) set that are not defined by a φ function in m. These variables are still live upon entry to m. All such variables are added to every set UEVAR(m, n) for each predecessor n of m.

For a program in SSA form, the recursive equation for LIVEOUT sets is rewritten as

$$\text{LIVEOUT}(n) = \bigcup_{m \in \text{succ}(n)} \text{UEVAR}(m, n) \cup \left(\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}\right). \quad (2)$$

In Fig. 1(c), $y_1 \in$ UEVAR(n_3, n_1) and $y_2 \in$ UEVAR(n_3, n_2). This yields the following live-out sets: LIVEOUT$(n_1) = \{y_1\}$ and LIVEOUT$(n_2) = \{y_2\}$.

VI. Chordal Graphs

This section summarizes relevant topics from the theory of chordal graphs. Historically, chordal graphs have also been called triangulated graphs and rigid circuit graphs. For a thorough treatment of the subject, the interested reader is referred to Golumbic's textbook on algorithmic graph theory [10, Ch. 4]. Fig. 2(a) and (b) shows respective examples of chordal and nonchordal graphs.

There are, in fact, three equivalent criteria for chordality. Undirected graph G is chordal if and only if:

1) G has no induced subgraph isomorphic to a k-hole;
2) G admits a perfect elimination order (PEO);
3) G is the intersection graph of a set of subtrees of a tree.

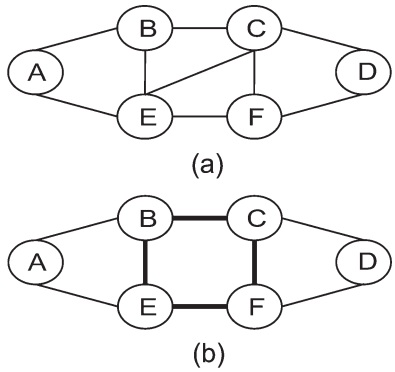

Fig. 2. (a) Chordal and (b) nonchordal graph. Four holes in (b) shown with bold edges.

Let $G = (V, E)$ be an undirected graph. Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic to one another if there exists a one-to-one bijection $g : V_1 \rightarrow V_2$, such that for all $u, v \in V_1$, $(u, v) \in E_1$ if and only if $(g(u), g(v)) \in E_2$.

A k-hole, defined by positive integer $k \geq 4$, is any graph isomorphic to $H = (V_O, E_O)$, where $V_O = \{v_0, v_1, \ldots, v_{k-1}\}$, and $E_O = \{(v_i, v_{(i+1)\%k}) | 0 \leq i \leq k-1\}$.

An elimination order σ of graph $G = (V, E)$ is a numbering scheme that orders the vertices. $\sigma(v) = i$ indicates that v is the ith element in the order. Once σ has been computed, vertices are renamed so that $\sigma(v_i) = i$.

Starting with an empty graph, G can be built by adding vertices in the order of σ. When v_i is added to G, all edges adjoining v_i to vertices in $\{v_1, \ldots, v_{i-1}\}$ are added to G as well. Let $G^{(i)} = (V^{(i)}, E^{(i)})$ represent the partially built graph after adding vertices $\{v_1, \ldots, v_i\}$; in other words, $G^{(i)}$ is the subgraph of G induced by $\{v_1, \ldots, v_i\}$.

Let $N(v)$ be the set of vertices adjacent to v in G, and let $N[v] = N(v) \cup \{v\}$. For $v \in V^{(i)}$, let $N^{(i)}(v) = N(v) \cap V^{(i)}$ and $N^{(i)}[v] = N^{(i)}(v) \cup \{v\}$.

Vertex v in G is simplical if $N(v)$ is a clique, a complete subgraph. σ is a PEO if v_i is simplical in $G^{(i)}$ for $i = 1, 2, \ldots, n$. In Fig. 2(a), $\langle B, C, E, FD, A\rangle$ is a PEO. On the other hand, $\langle A, B, C, D, E, F\rangle$ is not; E, the fifth vertex in the PEO, is not simplical since $N^{(5)}(E) = \{A, B, C\}$ is not a clique.

A PEO can be constructed for graph $G = (V, E)$ in $O(|V| + |E|)$ time using either lexicographic breadth first search (Lex-BFS) or a somewhat simpler and faster implementation of Lex-BFS called maximum cardinality search (MCS) [11]. Given a PEO, a chordal graph can be colored in $O(|V| + |E|)$ time [4]. Colors are assigned in PEO order: $\langle v_1, v_2, \ldots, v_n\rangle$. The only vertices that constrain the color that is assigned to v_i are those in $N^{(i)}(v_i)$. The first color not assigned to a vertex in $N^{(i)}(v_i)$ is given to v_i. If the colors are tested in some fixed order, a color will be found within $|N^{(i)}(v_i)| + 1$ tries.

VII. Optimal Register Sharing for SSA Form Programs

First, we introduce the concept of dominance. A node n_i in a CFG dominates node n_j (denoted n_i dom n_j) if every possible execution path from the entry node n_0 to n_j passes through

n_i. Trivially, n_0 dominates every other node in the CFG. The dominance relation satisfies the three properties:

reflexivity: n_i dom n_i

transitivity: n_i dom $n_j \wedge n_j$ dom $n_k \Rightarrow n_i$ dom n_k

antisymmetry: n_i dom $n_j \wedge n_j$ dom $n_i \Leftrightarrow n_i = n_j$.

For instructions, i_1 dom i_2 if i_1's basic block dominates i_2 or if i_1 precedes i_2 in the same basic block. For variables v_i and v_j, v_i dom v_j if the definition of v_i dominates the definition of v_j.

Theorems 1 and 2 are due to Budimlić *et al.* [8].

Theorem 1: If two variables u and v in a regular program interfere, then either u dom v or v dom u.

Theorem 2: If two variables v_1 and v_2 in a regular program interfere and the definition point of v_1 dom v_2, then v_1 is in the live-in set of the block in which v_2 is defined, or both variables are defined in the same block.

The live-in set in Theorem 2, denoted LIVEIN(n), is the set of variables that are live at n's entry point. Theorem 3 formally states that the interference graph for a regular program is chordal. A proof of Theorem 3 is given in the Appendix.

Theorem 3: Let $G = (V, E)$ be an interference graph generated from a regular program. Then G is a chordal graph.

At the International Workshop on Logic and Synthesis (IWLS) in June 2005, Brisk *et al.* [12] stated a slightly weaker version of Theorem 3—that G is a perfect graph. The proof in that paper is sufficient for chordal graphs, a subclass of perfect graphs. Brisk corrected this oversight in his talk.

In a technical report, also in June 2005, Hack [13] published Theorem 3 independently from Brisk *et al.* In fact, an earlier proof had been discovered in 2002 by Darte in a private conversation with Rastello and e-mail communication with Ferrante. Darte's proof used the subtree intersection characterization of chordal graphs (Definition 3). Darte's contribution was subsequently published in a technical report by Bouchez *et al.* [14] in August 2005.

The technical reports by Hack and Bouchez *et al.* discuss Theorem 3 in the context of register allocation in software compilers (e.g., [1], [19], and [20]), where the problem remains NP-Complete. Brisk *et al.* applied Theorem 3 to high-level synthesis, which has an analogous problem called register allocation in past literature (e.g., [3], [5], and [6]). The term "register sharing" is used here to refer to the problem in synthesis.

VIII. Computing Optimal Coloring Without Building Interference Graph

Given a CFG in SSA form, an optimal color assignment can be computed without explicitly constructing an interference graph. Let LIVEDEF(v) be the set of variables that are live at the point in the program just before v is defined. By Theorem 1, for every variable $u \in$ LIVEDEF(v), u dom v. A PEO can be constructed by creating an ordering σ, where $\sigma(v) > \sigma(u)$ for all $u \in$ LIVEDEF(v). For some variable v_i, such that $\sigma(v_i) = i$, LIVEDEF(v_i) is identical to clique $N^{(i)}(v_i)$ discussed in Section VI. Given LIVEDEF(v_i), the smallest color not assigned to a vertex in LIVEDEF(v_i) is then assigned to v_i.

```
Algorithm:  Chordal Color Assignment
Input:      CFG in SSA Form

1.   Let V be the set of SSA Variables
2.   For i ← 1 to |V|
3.       Free_Color[i] ← True
4.   For each basic block n, taken in dominance order
5.       LIVENOW ← LIVEOUT(n)
6.       For each instruction i* in b taken in reverse order
7.           KILLS(i*) ← φ
8.           For each variable u used by i*
9.               If u∉ LIVENOW
10.                  LIVENOW ← LIVENOW∪{u}
11.                  KILLS(i*) ← KILLS(i*)∪{u}
12.              EndIf
13.          EndFor
14.          Let v be the variable defined by i*
15.          LIVENOW ← LIVENOW - {v}
16.      EndFor
17.      // At this point, LIVENOW = LIVEIN(n)
18.      For each instruction i* in b taken in order
19.          Let v be the variable defined by i*
20.          j ← 1
21.          While( FreeColor[j] = False )
22.              j ← j + 1
23.          EndWhile
24.          Color[v] ← j
25.          FreeColor[j] ← False
26.          LIVENOW ← LIVENOW∪{v}
27.          For each variable u∈ KILLS(i*)
28.              FreeColor[Color[u]] ← True
29.              LIVENOW ← LIVENOW - {u}
30.          EndFor
31.      EndFor
32.  EndFor
```

Fig. 3. Pseudocode for optimal color assignment without building interference graph.

A pseudocode to perform color assignment without building an interference graph is given in Fig. 3. The dominator tree is traversed from the root toward the leaves, ensuring that each parent node in the tree is processed before all of its children. The instructions within each basic block n are traversed twice. First, n is traversed backward in order to determine which instructions represent the end of variable lifetimes. Let set KILLS(i^*) contain the variables whose lifetimes end at i^*. After computing the KILLS sets, a forward traversal of the list assigns colors to each variable. LIVEDEF(v) is replaced in Fig. 3 with a set LIVENOW that contains the variables that are live at each point in the CFG. Without KILLS sets, the forward traversal could not determine if each variable remains in LIVENOW or dies after each use.

Theorem 4 establishes the correctness of the algorithm in Fig. 3; a proof can be found in the Appendix.

Theorem 4: Let α be the order in which variables in a regular program are assigned colors by Line 24 in Fig. 3, and let $G = (V, E)$ be the interference graph. Then, σ is a PEO of G.

IX. Experimental Results

The Machine SUIF compiler [15] was used to test and evaluate the concepts and ideas presented in this paper. Programs are

TABLE I
RESULTS FOR REGISTER SHARING USING LINEAR SCAN ALLOCATION AND CHORDAL GRAPH COLORING

			Interference Graph		Number of Registers	
Benchmark	File	Procedure	|V|	|E|	Linear Scan	Chordal
176.gcc	combine.c	try_combine	7359	112759	66	24
176.gcc	combine.c	simplify_rtx	7018	41843	35	11
176.gcc	c-parse.c	yyparse	6260	61207	46	16
176.gcc	cse.c	fold_rtx	5879	67974	84	22
176.gcc	cse.c	cse_insn	6646	117368	109	33
176.gcc	expr.c	expand_expr	9753	63809	102	15
176.gcc	fold-const.c	fold	11135	83358	107	22
176.gcc	insn-recog.c	recog_5	5011	21197	44	6
176.gcc	jump.c	jump_optimize	6144	108078	71	29
186.crafy	option.c	Option	6193	7968	21	9
ghostscript	gdevstc.c	stc_put_params	5003	21464	24	9
pegwit	sha1.c	SHA1Transform	5663	46095	12	11

TABLE II
RUNTIME (IN SECONDS) FOR LIVENESS ANALYSIS AND THREE COLORING HEURISTICS (LS, CHO-1, CHO-2)

			CHO-1		
Procedure	Liveness Analysis	Linear Scan	Build Graph	PEO/Color	CHO-2
try_combine	6.59	0.208	1.48	3.25	0.0706
simplify_rtx	6.39	0.171	0.643	0.982	0.0714
yyparse	2.9	0.0719	0.883	1.35	0.04526
fold_rtx	5.94	0.213	1.11	1.25	0.158
cse_insn	2.61	0.147	0.934	1.61	0.0728
expand_expr	11.9	0.561	1.19	1.38	0.165
fold	23.2	0.511	1.5	1.55	0.181
recog_5	4.36	0.16	0.448	0.321	0.0285
jump_optimize	11.7	0.203	1.87	2.82	0.067
Option	3.14	0.0594	0.168	0.09471	0.02911
stc_put_params	4.2	0.147	0.408	0.297	0.131
SHA1Transform	0.237	0.141	0.45	0.769	0.034

converted to SSA form and an interference graph is constructed. Since this study focuses on register sharing (high-level synthesis) rather than register allocation (compilers), spilling and copy coalescing are not considered. Copies are folded during SSA construction and a pruned SSA form [16] is used in all experiments. The experiments were performed on a 3.00-GHz Intel Pentium IV Processor with 512-K cache and 1-G RAM, running Mepis Linux.

Register sharing based on chordal graph coloring is compared to register sharing based on linear scan allocation, which is summarized in Section X. The benchmarks used in this experiment are procedures written in C taken from SPEC INT 2000 [17] and Mediabench [18]. Each procedure in this experiment has an SSA-form interference graph with at least 5000 vertices. Results are not reported for smaller procedures. These procedures came from four benchmarks, namely: 1) 176.gcc; 2) 186.crafty; 3) ghostscript; and 4) pegwit.

Table I reports the effectiveness of register sharing using the two techniques. The first three columns summarize each benchmark: application, file, and function name. The next two columns, labeled $|V|$ and $|E|$, list the number of vertices and edges in the interference graph for the SSA form program. $|V|$ is the number of registers that would be allocated to the design in the absence of register sharing, one register per variable. The final two columns labeled show the number of registers allocated to the design after register sharing using both linear scan [1]–[3] and optimal chordal graph-coloring algorithms.

Table I clearly illustrates that chordal graph coloring outperforms linear scan. The difference in the number of registers allocated ranges from 1 (SHA1Transform) to 87 (expand_expr). On average, the difference in favor of chordal coloring was 42.83 registers per procedure. For smaller procedures, a linear scan is capable of producing (near-)optimal results. For procedures without control flow constructs such as conditional branches and loops, a linear scan is optimal.

The primary selling point of linear scan allocation is its runtime, not the quality of its results. Table II compares the runtime (in seconds) of linear scan to two different implementations of chordal graph coloring, an interference graph using the technique described in Section V computes a PEO using MCS and then colors the interference graph optimally [4]; CHO-2 uses the procedure in Fig. 3 to compute an optimal color assignment without building an interference graph.

From Table II, the runtime of liveness analysis, which is required by all three coloring techniques, dominates the total runtime cost of coloring. The respective runtimes of LS and CHO-2 post-liveness analysis are reported as is. The post-liveness analysis runtime of CHO-1 is broken into two components: the runtime of interference graph construction, including memory allocation, and the runtime of computing a PEO and

then coloring the chordal graph. Both LS and CHO-2 run much faster than CHO-1. Both components of CHO-1 run slower than LS and CHO-2 by at least one order of magnitude. CHO-2 runs faster than LS for all benchmarks.

The 12 interference graphs used in the experiments were much larger than the vast majority in both SPEC INT 2000 and Mediabench. The differences between CHO-2 and LS in terms of both solution quality and runtime are exacerbated here. For small benchmarks, the differences are negligible. For procedures without loops or branches, both LS and CHO-1/CHO-2 will produce optimal colorings.

X. Related Work

A. Register Allocation in Software Compilers

The first graph-coloring register allocator for compilers was developed by Chaitin *et al.* [19], [20]. This work included a proof that for every finite undirected graph G a CFG could be constructed such that liveness analysis would build an interference graph isomorphic to G. The CFG, however, was not strict, and the proof predated the discovery of an SSA form. If strictness was imposed on the CFG and the program was converted to SSA form, Theorem 3 would take precedence.

Numerous register allocators have been developed as modifications or improvements to Chaitin *et al.*'s allocator. For example, Pereira and Palsberg [21] have recently developed a Chaitin-style allocator that assigns colors using the MCS/PEO technique on both chordal and nonchordal interference graphs. Many successful register allocators have also been developed that do not use graph coloring.

Linear scan register allocation [1] emphasizes speed rather than solution quality. The program is represented as a linear list of instructions numbered $1, 2, \ldots, N$. Each variable v is represented by an interval $[i, j]$, where i is the smallest value such that there exists no $i' < i$, where v is live at i', and j is the largest value such that there exists no $j' > j$, where v is live at j'. Given a variable v with interval $[i, j]$, there may be many subintervals within $[i, j]$ where v is not live. An optimal color assignment for the linear scan interference graph is often suboptimal for exact interference graphs.

B. Translation Out of SSA Form

The process of translating a program out of an SSA form is a necessary step for software compilers. To eliminate the φ function $y \leftarrow \varphi(\ldots, x, \ldots)$, there are two options: insert a copy $y \leftarrow x$, or merge x and y into one variable in the post-SSA program. x and y can only be merged together if they do not interfere.

Cytron *et al.* [22] and Briggs *et al.* [23] inserted copies for all variables and relied on the coalescing phase of a Chaitin-style register allocator to remove as many copies as possible. Rastello *et al.* [24] showed that translation out of an SSA form is NP-Complete and tried to minimize the number of copies inserted while also satisfying architecture-imposed naming constraints on variables in the post-SSA program. Budimlić *et al.* [8] used Theorems 1 and 2 to develop a heuristic with an almost-linear time complexity that tried to minimize the number of copies inserted during translation out of SSA.

C. Implications of Theorem 3 for Software Compilers

Theorem 3 yields an optimal $O(|V| + |E|)$-time algorithm for register sharing in high-level synthesis for CFGs in SSA form. Theorem 3 does not, however, yield an optimal SSA-based polynomial-time solution for register allocation in compilers. In a compiler, a register allocator must determine which variables to store in registers and which to spill to memory. If k is the number of registers in the target architecture, then the subgraph induced by the subset of variables stored in registers must be k colorable. All variables not in the induced subgraph would be spilled to memory instead. Unfortunately, the problem of determining the largest k colorable induced subgraph is NP-Complete for chordal graphs [25]. Moreover, once spilling decisions have been made, the placement of spill code is NP-Complete [26]. Finally, minimizing the number of copies inserted during SSA deconstruction is NP-Complete as well [24].

XI. Conclusion

The problem of register sharing on a CFG in SSA form has been shown to have a polynomial-time solution based on chordal graph coloring. By sidestepping the construction of the interference graph, optimal register sharing can run faster than linear scan allocation. Register sharing will reduce the overall area of the design and increase the utilization of registers. If resource R is connected to k registers $r_1, r_2, \ldots r_k$ that are consolidated into a single register r via register sharing, then the interconnect may be reduced as well.

Techniques based on translation out of SSA form should be investigated to reduce the number of multiplexers required to synthesize a data path. Consider φ function $y \leftarrow \varphi(x_1, x_2, x_3, x_4)$. If $\{x_1, x_2, x_3, x_4\}$ are assigned to the same register, then no multiplexer is necessary. If $\{x_1, x_2\}$ and $\{x_3, x_4\}$ are assigned to two registers, then a two-input multiplexer is necessary; if $\{x_1\}$, $\{x_2\}$, $\{x_3\}$, and $\{x_4\}$ are assigned to four different registers, then a four-input multiplexer is required. This problem is most likely NP-Complete due to its similarity to copy minimization during translation out of SSA form.

Appendix

Proof of Theorem 3: Let $V' \subseteq V$, $|V'| = k \geq 4$. We show that G', the subgraph of G induced by V', is not isomorphic to a k hole. Assume to the contrary that G' is isomorphic to a k hole, i.e., $V' = \{v_0, v_1, \ldots, v_{k-1}\}$ and $E' = \{(v_i, v_{(i+1)\%k}) | 0 \leq i \leq k-1\}$. Let us assign directions to the edges in E' in accordance with dominance. By Theorem 1, for every edge $(v_i, v_{(i+1)\%k}) \in E'$, either v_i dom $v_{(i+1)\%k}$ or $v_{(i+1)\%k}$ dom v_i. After assigning directions, all edges will be written such that $(v_i, v_{(i+1)\%k}) \in E'$ indicates that v_i dom $v_{(i+1)\%k}$. Henceforth, G' is directed.

First, we argue that G' must be acyclic. Since G' is a hole, the only cycle would be the sequence $\langle v_0, v_1, \ldots, v_k, v_0 \rangle$, its

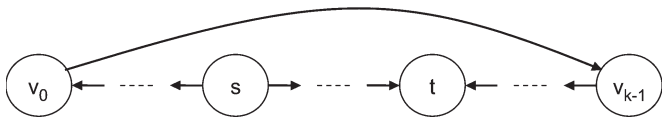

Fig. 4. k hole represented as DAG with source s and sink t. Both of t's neighbors are proven to interfere; however, there can be no edge between t's neighbors in k hole, a contradiction.

reverse, or a circular shift thereof. Without loss of generality, let v_0 dom v_1; however, path $\langle v_1, \ldots, v_k, v_0 \rangle$ coupled with the transitivity of the dominance relation indicates that v_1 dom v_0, violating antisymmetry. Therefore, G' must be acyclic.

In a directed graph, $N(v)$ is divided into incoming and outgoing adjacent vertices. $N_{\text{in}}[v] = \{u|(u,v) \in E'\}$ and $N_{\text{out}}[v] = \{w|(v,w) \in E'\}$. Since the original undirected graph G is a hole, every vertex v satisfies the following property: $|N_{\text{in}}[v]| + ||N_{\text{out}}[v]| = 2$. A source is defined to be a vertex s such that $|N_{\text{in}}[s]| = 0$. A sink is defined to be a vertex t such that $|N_{\text{out}}[t]| = 0$. It is a fundamental result from graph theory that every DAG has at least one source and at least one sink.

Let $s = v_i$ and $t = v_j$ be a respective source and sink in G'. W.L.O.G., suppose that $i < j$. If $j > i$, simply reverse all of the edges in G' and swap s and t. Antisymmetry ensures that the same vertex cannot be both a source and a sink in G'.

We do not know whether s and t are the unique source and sink, or if there are others. If s and t are unique, then there exist two directed paths from s to t: $P_1 = \langle s, v_{i+1}, v_{i+2}, \ldots, t \rangle$ and $P_2 = \langle s, v_{i-1}, v_{i-2}, \ldots v_0, v_{k-1}, v_{k-2}, \ldots t \rangle$, illustrated in Fig. 4.

We show by induction that P_1 is a directed path in G. For the basis, $\langle s, v_{i+1} \rangle$ is trivially a directed path since s is a source. Now, let $\langle s, v_{i+1}, \ldots, v_{i+\alpha} \rangle$, where $1 \leq \alpha \leq j - i - 1$ be a directed path from s to $v_{i+\alpha}$. Now, consider $v_{i+(\alpha+1)}$.

Assume to the contrary that $(v_{i+(\alpha+1)}, v_{i+\alpha})$ defines the direction of the edge between $v_{i+(\alpha+1)}$ and $v_{i+\alpha}$; consequently, $v_{i+(\alpha+1)}$ dom $v_{i+\alpha}$. Let n be the basic block that defines $v_{i+(\alpha-1)}$. By Theorem 2, either $v_{i+\alpha} \in \text{LIVEIN}(n)$ or $v_{i+\alpha}$ is defined in n.

By the induction hypothesis, the direction of edge $(v_{i+(\alpha-1)}, v_{i+\alpha})$ has already been resolved; consequently, $v_{i+(\alpha+1)}$ dom $v_{i+\alpha}$. Once again, Theorem 2 shows that either $v_{i+(\alpha-1)} \in \text{LIVEIN}(n)$ or $v_{i+(\alpha-1)}$ is defined in n. In either case, $v_{i+(\alpha-1)}$ and $v_{i+(\alpha+1)}$ must both be live at the definition point of $v_{i+\alpha}$ in n, and $v_{i+(\alpha-1)}$ and $v_{i+(\alpha+1)}$ interfere. Since a hole must contain at least four vertices, and edges $(v_{i+(\alpha-1)}, v_{i+\alpha})$ and $(v_{i+\alpha}, v_{i+(\alpha+1)})$ are known to exist, the existence of edge $(v_{i+(\alpha-1)}, v_{i+(\alpha+1)})$ would violate the assumption that G' is a hole, a contradiction. Consequently, $(v_{i+\alpha}, v_{i+(\alpha+1)})$ must be the direction of the edge, ensuring that $\langle s, v_{i+1}, \ldots, v_{i+\alpha}, v_{i+(\alpha+1)} \rangle$ is a directed path from s to $v_{i+(\alpha+1)}$.

This establishes that P_1 is a directed path from s to t. An identical argument proves that P_2 is also a directed path from s to t. Together, the existence of directed paths P_1 and P_2 ensures that there is exactly one source and one sink in G'. Now, consider sink t. Since t is a sink, $N_{\text{in}}[t] = \{v_{j-1}, v_{j+1}\}$. Consequently, v_{j-1} dom t and v_{j+1} dom t. If n is the basic block in which t is defined, then $v_{j-1} \in \text{LIVEIN}(n)$ or v_{j-1} is defined in n; likewise, $v_{j+1} \in \text{LIVEIN}(n)$ or v_{j+1} is defined in n. Both v_{j-1} and v_{j+1} interfere at the definition point of v_j. Once again, a hole known to contain edges (v_{j-1}, v_j) and (v_j, v_{j+1}) cannot also contain an edge (v_{j-1}, v_{j+1}). This contradicts the assumption that G' is isomorphic to a k-hole. ■

Proof of Theorem 4: Consider a basic block n. Let DEFS$[n]$ be the set of variables defined in n. For $u \in$ LIVEIN(n), $v \in$ DEFS$[n]$, $\sigma(u) < \sigma(v)$ since basic blocks are processed in dominance order. For $v, w \in$ DEFS$[n]$, $\sigma(v) < \sigma(w)$ if and only if v is defined before w in n since Line 24 occurs within a forward traversal of n. Then, for each variable $x \in$ LIVEDEF$[v]$, $\sigma(v) > \sigma(x)$.

Now, consider interference edge $(u, v) \in E$. By Theorem 2, either (1) u dom v and $u \in$ LIVEDEF$[v]$ or (2) v dom u and $\sigma(v) < \sigma(u)$. Let $G' = (V', E')$ be the subgraph of G induced by $V' = \{(u,v) \in E|\sigma(u) < \sigma(v)\}$. Then, $(u, v) \in E'$ if and only if $u \in$ LIVEDEF$[v]$. Since LIVEDEF$[v]$ is a clique, v is simplical in G' and σ is a PEO of G. ■

ACKNOWLEDGMENT

The following individuals provided insightful comments on this work: D. Berlin, A. Darte, S. Hack, J. Palsberg, F. M. Q. Pereira, F. Rastello, L. Stok, and the anonymous reviewers.

REFERENCES

[1] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999.

[2] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *Proc. 8th Workshop Design Automation*, Atlantic City, NJ, Jun. 1971, pp. 155–169.

[3] F. J. Kurdahi and A. C. Parker, "REAL: A program for register allocation," in *Proc. 24th Design Automation Conf.*, Miami Beach, FL, 1987, pp. 210–215.

[4] 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, Jun. 1972.

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

[6] L. Stok, "Transfer-free register allocation in cyclic data flow graphs," in *Proc. Eur. Conf. Design Automation*, Brussels, Belgium, Mar. 1992, pp. 181–186.

[7] A. Kaplan, P. Brisk, and R. Kastner, "Data communication estimation and reduction for reconfigurable systems," in *Proc. 40th Design Automation Conf.*, Anaheim, CA, Jun. 2003, pp. 616–621.

[8] Z. Budimlić, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. W. Reeves, "Fast copy coalescing and live-range identification," in *Proc. ACM SIGPLAN Conf. Programming Language Design Implementation*, Munich, Germany, Jun. 2002, pp. 25–32.

[9] K. D. Cooper and L. Torczon, *Engineering a Compiler*. San Francisco, CA: Morgan Kaufmann, 2004.

[10] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics*, vol. 57. Amsterdam, The Netherlands: North Holland, 2004.

[11] 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, Aug. 1984.

[12] P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh, "Polynomial time graph coloring register allocation," in *Proc. IWLS*, Lake Arrowhead, CA, Jun. 2005, pp. 447–454.

[13] S. Hack, "Interference graphs of programs in SSA-form," Universität Karlsruhe, Karlsruhe, Germany, Jun. 2005. (ISSN 1432-7864).

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

[15] M. D. Smith and G. Holloway, *An Introduction to Machine SUIF and Its Portable Libraries for Optimization*. [Online]. Available: http://www.eecs.harvard.edu/hube/software/nci/overview.html
[16] J.-D. Choi, R. Cytron, and J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," in *Proc. 18th Annu. ACM Symp. Principles Programming Languages*, Orlando, FL, Jan. 1991, pp. 55–66.
[17] *Standard Performance Evaluation Consortium*. [Online]. Available: http://www.spec.org/cpu2000/
[18] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. 30th Int. Symp. Microarchitecture*, Research Triangle Park, NC, Dec. 1997, pp. 330–335.
[19] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proc. ACM SIGPLAN Symp. Compiler Construction*, Boston, MA, Jun. 1982, pp. 98–105.
[20] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Comput. Lang.*, vol. 6, no. 1, pp. 47–57, 1981.
[21] F. M. Q. Pereira and J. Palsberg, "Register allocation via coloring of chordal graphs," in *Proc. 3rd Asian Symp. Programming Languages Systems*, Tsukuba, Japan, Nov. 3–5, 2005, pp. 315–329.
[22] 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 Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.
[23] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, "Practical improvements to the construction and destruction of static single assignment form," *Softw.—Pract. Exp.*, vol. 28, no. 8, pp. 859–881, Jul. 1998.
[24] F. Rastello, F. de Ferriere, and C. Guillon, "Optimizing translation out of SSA using renaming constraints," in *Proc. Int. Symp. Code Generation Optimization*, Palo Alto, CA, Mar. 21–24, 2004, pp. 265–276.
[25] M. Yannakakis and F. Gavril, "The maximum k-colorable subgraph problem for chordal graphs," *Inf. Process. Lett.*, vol. 24, no. 2, pp. 133–137, Jan. 1987.
[26] M. Farach and V. Liberatore, "On local register allocation," in *Proc. 9th Annu. ACM SIAM Symp. Discrete Algorithms*, San Francisco, CA, Jan. 25–27, 1998, pp. 564–573.

Philip Brisk received the B.S. and M.S. degrees in computer science from the University of California, Los Angeles, in 2002 and 2003, respectively.

He is currently with the Embedded and Reconfigurable Systems Laboratory, Department of Computer Science, University of California. His research interests include compilers, synthesis, and code compression.

Foad Dabiri (S'05) received the B.S. degree in electrical engineering from the Sharif University of Technology, Tehran, Iran, in 2003, and the M.S. degree in computer science from the University of California, Los Angeles, CA, in 2005.

He is currently with the Embedded and Reconfigurable Systems Laboratory, Department of Computer Science, University of California. His research interests include algorithm design and analysis for embedded systems with emphasis on optimization algorithms targeting performance and power.

Roozbeh Jafari (S'99) received the B.Sc. degree in electrical engineering from the Sharif University of Technology, Tehran, Iran, in 2000, the M.Sc. degree in electrical engineering from the State University of New York, Buffalo, NY, in 2002, and the M.S. degree in computer science from the University of California, Los Angeles, in 2004.

He was with IBM, Endicott, NY, developing the IBM TestBench tool designed for very large scale integration (VLSI) testing. He is currently with the Embedded and Reconfigurable Systems Laboratory, Department of Computer Science, University of California. His research interests include networked embedded system design and reconfigurable computing with emphasis in medical/biological algorithms and applications.

Majid Sarrafzadeh (S'82–M'82–SM'91–F'96) received the B.S., M.S., and Ph.D. degrees in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1982, 1984, and 1987, respectively.

From 1987 to 2000, he was a member of the Faculty at the Department of Electrical and Computer Engineering, Northwestern University. In 2000, he joined the Department of Computer Science, University of California, Los Angeles. He has collaborated with many industries in the past 15 years including IBM and Motorola and many computer-aided design (CAD) industries and was the architect of the physical design subsystem of Monterey Design Systems' main product. He is a co-founder of Hier Design, Inc. He has published approximately 250 papers, is the coeditor of the book *Algorithmic Aspects of VLSI Layout* (World Scientific, 1994), and is the coauthor of the books *An Introduction to VLSI Physical Design* (McGraw Hill, 1996) and *Modern Placement Techniques* (Kluwer, 2003). His recent research interests lie in the area of embedded and reconfigurable computing, very large scale integration CAD, and algorithm design and analysis.

Dr. Sarrafzadeh is on the editorial board of the VLSI Design Journal, an Associate Editor of the ACM Transaction on Design Automation, and an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN.