A Polynomial Time Exact Solution to the Bit-Aware Register Binding Problem

Michael Canesche
UFMG
Brazil
michael.canesche@dcc.ufmg.br

Ricardo Ferreira
UFV
Brazil
ricardo@ufv.br

José Augusto Nacif
UFV
Brazil
jnacif@ufv.br

Fernando Magno Quintão Pereira
UFMG
Brazil
fernando@dcc.ufmg.br

Abstract

Finding the minimum register bank is an optimization problem related to the synthesis of hardware. Given a program, the problem asks for the minimum number of registers plus their minimum size, in bits, that suffices to compile said program. This problem is NP-complete; hence, usually solved via heuristics. In this paper, we show that this problem has an optimal solution in polynomial time, as long as swaps can be inserted in the program to move variables across registers. This observation sets a lower bound to heuristics that minimize the size of register banks. We have compared the optimal algorithm with two classic heuristics. Our approach uses, on average, 6 to 10% less bits than that previous work.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Compilers**.

Keywords: Register allocation, Optimization, Polynomial, High-Level Synthesis, Bit-width

ACM Reference Format:
Michael Canesche, Ricardo Ferreira, José Augusto Nacif, and Fernando Magno Quintão Pereira. 2022. A Polynomial Time Exact Solution to the Bit-Aware Register Binding Problem. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3497776.3517773

1 Introduction

Bit-width minimization, in the context of high-level synthesis, has been historically solved by algorithms that perform *register assignment* in compilers. This problem consists in finding the minimum number of registers necessary to compile a program. Register assignment is NP-complete in general, as it subsumes graph coloring [5]. However, since 2005, it is known that register assignment has a polynomial-time solution when restricted to programs in Static Single Assignment (SSA) form [9], because SSA-form programs have chordal interference graphs [1, 16, 24]. Any program can be converted to SSA form in polynomial time. The transformed program will not require more registers than its original counterpart—often requiring less[1]. This paper brings some of these ideas to the hardware-synthesis domain.

Bit-Aware Resource Allocation. Those early results, related to register assignment in SSA-form programs, concern traditional compilation, in which registers have all the same length. However, in hardware synthesis, nothing hinders the synthesizer from using resources (registers, adders, interconnects, etc) of different bit widths. In this setting, register assignment is NP-complete, even for SSA-form programs, and even if only two sizes of registers are admissible [12, 20]. As an example, register assignment is NP-complete for x86, where two registers of eight bits can fit into one register of 16 [26]. In 2008, Pereira and Palsberg [26] showed that register assignment with registers of two sizes has a polynomial solution in a program representation called *Elementary Form*. In this representation, variables can be copied from one register to another at every point in between two consecutive assembly instructions. Although showing promise for hardware design, Pereira and Palsberg's result is too constrained to be of much use in this domain: only two sizes of registers are allowed, and one of these sizes must be at least twice as large as the other. In this paper, we go beyond that.

[1]Notice that it is not known if an optimal solution to register assignment, found for an SSA-form program, can be projected back onto the original program [25]. Otherwise, we would have that P=NP!

The Contribution of this Paper. This paper revisits the *register binding problem*, a variation of register assignment in which neither the number of registers nor their bit widths are fixed. As we explain in Section 2, the solver is allowed to assume the existence of an infinite supply of bits to construct a register bank. Section 3 shows that optimal register binding admits a polynomial-time solution in Pereira and Palsberg's elementary representation. By "optimal", we mean that the total size of registers, in bits, is minimized. Any program can be converted into the elementary format in polynomial time, and synthesis in that representation never consumes more registers. Nevertheless, optimality has a price, paid in terms of copy and swap instructions necessary to build the elementary format. The problem of minimizing these instructions is NP-complete, as already demonstrated in the original formulation of elementary form [26].

Summary of Results. We have implemented a proof-of-concept resource allocator in the LLVM compiler [19]. Section 4 compares our optimal algorithm with two approximate solutions to register binding. The first is a heuristic conceived by Cong et al. [7] to do *local* register binding. By "local", we mean that the problem is restricted to programs without branches. The second heuristic is the greedy algorithm used to solve weighted-coloring in interval graphs [11]. Interval graph coloring has been shown to solve register binding optimally [2] on SSA-form programs without branches, assuming that variables have the same size. We emphasize that the algorithm proposed in this paper is "global": it works for programs with branches. Nevertheless, local heuristics can still be applied onto whole programs via a process called "linearization", as explained in Section 4.

We have run experiments on all the 492 functions taken from the 15 programs available in the LLVM version of the MiBench suite [15]. This evaluation shows a number of facts. First, there is little room for improving on typical heuristics. The optimal solution has improved over Cong's heuristic in 190 cases. Improvements over greedy coloring happened in 145 cases. Notice that all these heuristics outperform by a wide margin typical register allocators based on the coloring of interval graphs that are not bit-aware. Second, to achieve optimality, we had to insert, on average, one copy or swap instruction for each 81 instructions in the original programs. Nevertheless, although offering slight practical improvement, we claim that the existence of an optimal algorithm is, in itself, a contribution to the high-level synthesis community, as it provides a lower bound against which any new approximation algorithm can be tested.

Synthesis. The hardware synthesizers that we are familiar with (`Intel HLS` [18], Xilinx's `Vivado` [38] and `LegUp` [4]) do not support bit-aware resource binding. Therefore, we cannot currently map the results produced by our LLVM backend directly onto logic gates. However, to demonstrate that our approach can be practical, we have manually coded a few programs compiled with the different heuristics evaluated in this paper in Verilog [37]. The extended version of this paper [23] shows that: (i) the number of bits found by the register binding algorithm can be preserved throughout the process of hardware synthesis for FPGA; and (ii) the exact algorithm outperforms, at least for the tried examples, heuristics previously used in this domain.

2 Overview

This paper is about the *Local Register Binding* problem, which Definition 2.1 states. Definition 2.1 uses a number of terms that are standard in the jargon of compiler engineers. For the sake of completeness, we revise them. A *straight-line program* is a program without branches. An SSA-form program is a program in which every variable is defined at only one site, and every definition dominates every use. Henceforth, we assume that every program meets the SSA property. A program point is any point in between two consecutive instructions. We write $v \in P$ to indicate that variable v is mentioned within SSA-form program P. A variable v is said to be *alive* at a program point p of an SSA-form program if there exists an instruction ι_a that defines v; an instruction ι_b that uses v; a path from ι_a to p; and a path from p to ι_b. This definition is sound for SSA-form programs, as variable names cannot be redefined. The *live range* of a variable is the set of program points where the variable is alive. Two live ranges overlap if they share a common program point.

Definition 2.1 (Local Register Binding–LRB)**. Input:** a program P in static single assignment form, a table R that maps variables in P to a width, in bits, and two integers, I_r and I_b. **Output:** a mapping A of every variable $v \in P$ to a register r_i, $1 \leq i \leq R$, that meets the two constraints:

1. If the live range of $v_a \in P$ and $v_b \in P$ overlap, then $A[v_a] \neq A[v_b]$.
2. Let S_p be the set of variables alive at program point p, and A_p be the registers assigned to variables in S_p:
   a. $|S_p| = |A_p| \leq I_r$.
   b. If σ_p is the sum of the widths of registers in A_p, then $\sigma_p \leq I_b$.

Example 2.2. Figure 1 (Left) shows a straight-line program with six instructions (thus, seven program points). Figure 1 (Right) shows the instance of the local register binding problem posed by that program. Each box denotes one bit; thus, variable a requires a register of at least five bits. Given this example, we can simplify Definition 2.1 stating, rather informally, that a solution to LRB consists of packing all the colored boxes as tight as possible, without allowing overlaps.

2.1 Brisk et al.'s Solution to LRB

If we assume that every variable in the target program has the same bit width ($\forall v_a, v_b, R(v_a) = R(v_b)$), then LRB can be solved in polynomial time using, for instance, Brisk et al.

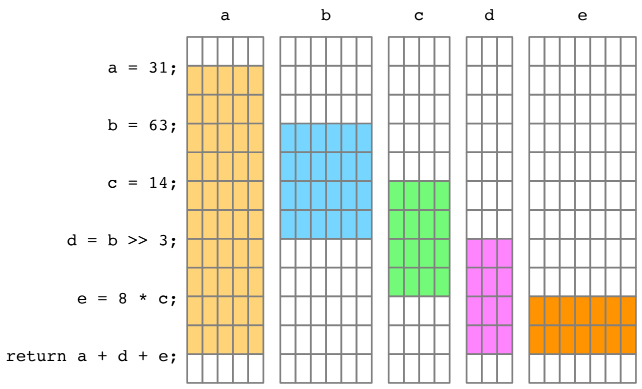


Figure 1. A straight-line program written in C, plus the live ranges of its variables. The Local Register Binding problem consists in packing these live ranges as tight as possible, without overlaps.

[2]'s method. This technique uses interval graph coloring to assign registers to variables in a straight-line program: a list of *free* (F) and *taken* (T) registers is updated at each program point, in ascending order. Whenever the last use of a variable is visited, its register is returned to the *free* list. Whenever a variable is allocated to a register, this register goes to the *taken* list. The chromatic number of interval graphs can be found in polynomial time [13]; thus, assignment is optimal.

It is possible to adapt Brisk et al.'s method to incorporate bit widths. Said adaptation requires us to keep a map W that associates registers with their widths (which might increase while the program is scanned). Whenever necessary to allocate a new color to the interval that corresponds to variable v, it suffices to apply one of the three actions below, in order:

1. If there exists a free register r, such that $W(r) \geq R(v)$, assign it to v;
2. Else, if the list of free registers is not empty, then pick the register r whose width $W(r)$ approximates $R(v)$ from below, and assign it to v. Make $W(r) = R(v)$;
3. Else, create a new register r, such that $W(r) = R(v)$ and assign it to v.

In any of these three cases above, the register r that is assigned to variable v ends up in the list of taken registers. Henceforth, we shall call this solution the *Bit-Aware Brisk (Bab) Approach*. Figure 2 provides its pseudocode.

Example 2.3. Figure 3 (a) shows the solution produced by the bit-aware version of Brisk et al.'s approach on the program in Figure 1. This solution led to a bank containing three registers whose bit widths add up to 18 bits.

2.2 Cong et al.'s Solution to LRB

Cong et al. [7] have proposed a heuristic to solve the local register binding problem, which generalizes previously documented techniques [35]. Contrary to Brisk et al.'s method, this algorithm was designed to accommodate variables with different bit widths. Cong et al.'s algorithm is also greedy; however, instead of iterating over the instructions in the program, it iterates over the widths of the variables defined by those instructions, in descending order. The algorithm packs into the same register variables of similar bit width.

```
Input:
  Program P
  Map<Var, int> R # Map variables to bitwidths
Data structures:
  List<Reg> F # List of free registers
  List<Reg> T # List of taken registers
Output:
  Map<Reg, int> W # Map registers to bitwidths
  Map<Var, Reg> R # Map variables to registers
where:
  for Instruction i in P do
    for Var v last used at i do
      add R(v) to F
    for Var v defined at i do
      if ∃r' in F, such that W(r') ≥ R(v) then
        let r in F minimize W(r) - R(v)
      else if F ≠ ∅ then
        let r in F minimize R(v) - W(r)
        W(r) = R(v)
      else
        create r
        W(r) = R(v)
      add r to T
      R(v) = r
    end
  end
```

Figure 2. A bit-aware adaptation of the method originally proposed by Brisk et al. [2] to synthesize locations for the variables in a program.

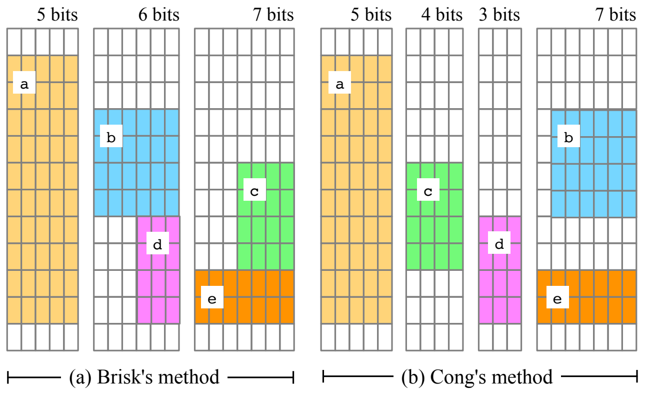


Figure 3. Different allocations produced for the program in Figure 1(a). (a) Allocation produced by the bit-aware adaptation of Brisk et al.'s approach. (b) Allocation produced by Cong et al. [7]'s approach.

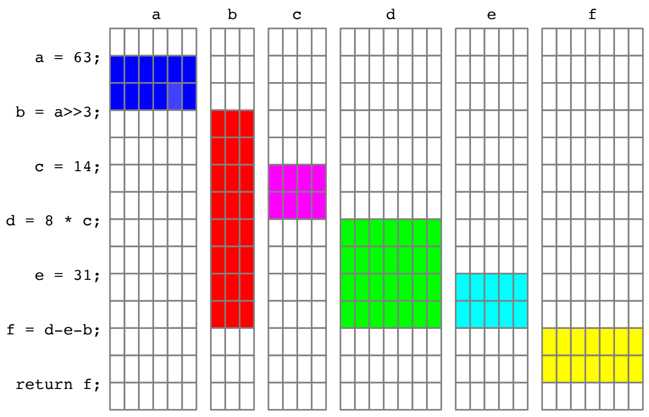
```
Input:
  Program P
  Map<Var, int> R # Map variables to bitwidths
Data structures:
  Queue<Var> Q # Priority queue of variables
  List<Var> A # List of active variables
  Set<Var> U # Set of variables to be revisited
Output:
  Map<Reg, int> W # Map registers to bitwidths
  Map<Var, Reg> R # Map variables to registers
where:
  for Var v in P do
    add v to Q
  end
  while Q ≠ ∅ do
    let v be the highest priority variable from Q
    remove v from Q
    add v to A
    create r
    W(r) = R(v)
    for Var u in Q do
      remove u from Q
      if u does not overlap any variable in A then
        add u to A
        R(u) = r
      else
        add u to U
    for Var u in U do
      add u to Q
    end
    A = U = ∅
  end
```

Figure 4. The method proposed by Cong et al. [7] to synthesize locations for the variables in a program.

Figure 4 shows Cong et al.'s method. This algorithm uses a priority queue Q, sorted by the width of variables in *descending order*. While Q is not empty, it removes the top variable v from it. A new register r is created for v. At this point, the algorithm tries to assign every other variable, in the descending order of bit widths, to r. This allocation is possible as long as no variable assigned to r interferes with each other. Example 2.4 illustrates how this algorithm works.

Example 2.4. Figure 3 (b) shows the solution produced by Cong *et al.*'s method when applied onto the program earlier seen in Figure 1. This solution led to a final register bank with four registers spanning 19 bits.

Neither Brisk et al.'s nor Cong et al.'s solution to the LRB problem subsume each other. The former approach always delivers the smallest number of registers necessary to execute a program. However, once made bit-aware, it might yield suboptimal solutions, just like Cong's technique. For the program in Figure 1, Bab outperforms Cong's in terms of the number of registers (3 vs. 4) and in the number of bits (18 vs. 19). Example 2.5 shows an example where Cong et al.'s method is better. Nevertheless, the solution proposed in this paper never uses more bits nor registers than either of these two previous approaches.



```
a = 63;
b = a>>3;
c = 14;
d = 8 * c;
e = 31;
f = d-e-b;
return f;
```

Figure 5. A new instance of local register binding in which Bab yields a worst solution than Cong's algorithm.

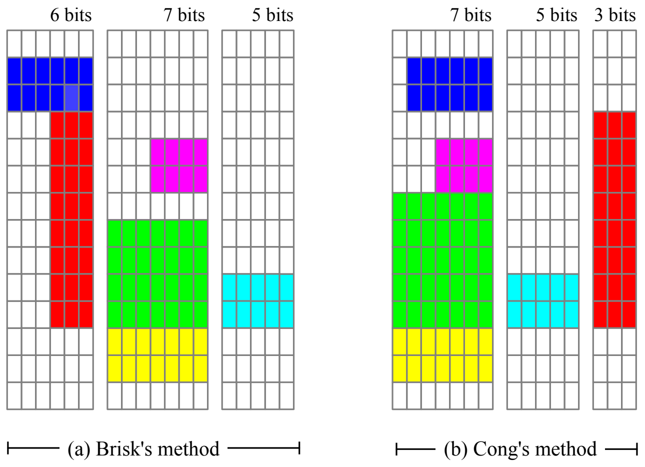


Figure 6. Two solutions for the LRB instance seen in Figure 5.

Example 2.5. Figure 5 shows an instance of the local register binding problem in which Cong's technique leads to a register bank with less bits than Bab. Figure 6 shows the solutions found by these two techniques. In this example, both methods produced a register bank with three registers. However, Cong et al.'s approach uses 3 + 5 + 7 bits, whereas our adaptation of Brisk et al.'s algorithm uses 5 + 6 + 7 bits.

3 Optimal Register Assignment with Swaps

Figure 7 provides an overview of the algorithm that will be presented in this section. This algorithm is formed by three parts: conversion to the elementary format (Section 3.1); register binding (Section 3.2) and elimination of the elementary format (Section 3.3). These parts are independent, meaning that the implementation of either one of them bears no impact on the implementation of the other.

3.1 Conversion to the Elementary Format

Register assignment has a polynomial-time solution in *elementary-form programs.* A program is in the elementary format if the live range of any variable encompasses only one program point. To ensure this property, variables are renamed between consecutive instructions. The next example shows how a program looks when converted into this format.

Example 3.1. Figure 8 shows a program in elementary format. Copies in boxes are abstract instructions used to rename variables across program points.

The *copies* mentioned in Example 3.1 are called *Parallel Copies* [26]. A notation like $(a_3, b_3) \Leftarrow (a_2, b_2)$ means that the values of a_2 and b_2 are moved into a_3 and b_3 in parallel. Thus, the values on the right side are read before being placed into the locations on the left side. Parallel copies do not exist in an actual program. Rather, they indicate that the actual location of a variable is not fixed between program points. Nevertheless, they must be implemented, once variables are mapped into registers. It is possible to implement parallel copies using a combination of simple copies and register swaps without the need for auxiliary locations [27]. The extended version of this paper shows how to implement parallel copies in hardware [23].

```
Input:
  Program P
  Map<Var, int> W # Map variables to bitwidths
Data structures:
  List<int[]*int[]*int[]> E # Elementary graphs
  List<List<Var ↦ Reg>> C # Pointwise reg mapping
Output:
  Map<Reg, int> W # Map registers to bitwidths
  Map<Var, Reg> R # Map variables to registers
where:
  for Instruction*Instruction (i, succ_i) in P do
    # See Section 3.1:
    (Q1, Q2, Qx) = elementary_graph(i, succ_i)
    add (Q1, Q2, Qx) to E
  end
  for (Q1, Q2, Qx) in E do # See Figure 10.
    C_w = MSUML(Q1, Q2, Qx, W) # Definition 3.2.
    P = map(Q1, Q2) # Assign to same r vars v1 &
    add P to C      # v2 if their colors overlap.
  end # In what follows, refer to Theorem 3.6:
  for i in length(E)-1 do # See Section 3.3.
    (Q1, Q2, Qx) = E[i];  # Sort and merge regs
    (Q1', Q2', Qx') = E[i+1]; # at join points.
    (S1,S2) = MSUMLL(Q2+QX, Q1'+Qx', C[i], C[i+1]);
    # The weight of reg is max at the two points:
    updt_reg_weights(W, R, S1, S2);
    par_copies(W, R, S1, S2); # [Pereira and Palsberg'09]
  end
```

Figure 7. Overview of the algorithm proposed in this paper.

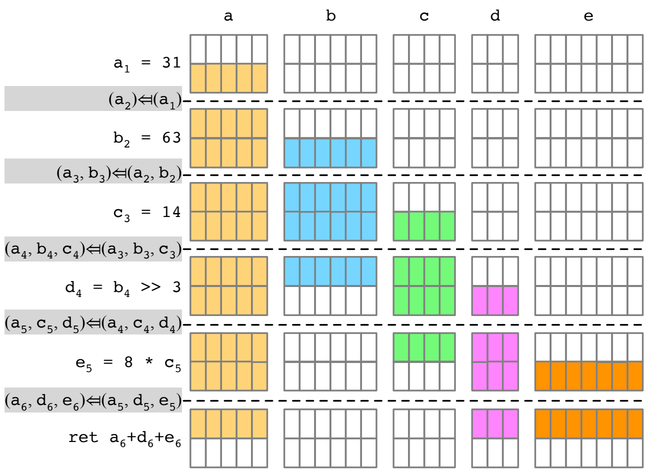


Figure 8. The program in Figure 1, converted into Elementary Form.

3.2 Weighted Graph Coloring

Local Register Binding becomes straightforward when performed in the elementary format. In this representation, LRB is equivalent to solving multiple instances of the *Weighted Graph Coloring Problem* on *elementary graphs.* The union of two cliques forms elementary graphs. Because this problem is of capital importance to this presentation, we formalize it in Definition 3.2 and illustrate it in Example 3.3.

Definition 3.2 (WGCE). **Input:** a weighted elementary graph $G = (Q_1, Q_2, Q_x, W)$, where $Q_1 \cup Q_x$ and $Q_2 \cup Q_x$ form cliques, and $W : V \mapsto N$ is a map that associates vertices with positive integer weights, plus an integer K. **Output:** a weighted coloring $C_w : V \mapsto [N \times N]$ that associates vertices with positive integer intervals, such that for any two vertices v_i and v_n in the same clique, $C_w(v_i) \cap C_w(v_j) = \emptyset$, and the sum of all the intervals is less than K.

Example 3.3. Figure 9 shows a solution to the Weighted Coloring Problem on six elementary graphs. These graphs represent the interferences between the variables used in the program from Example 3.1. For instance, the fourth graph, corresponding to instruction $d_4 = b_4 >> 3$, is formed by $Q_1 = \{b_4\}$, $Q_2 = \{d_4\}$ and $Q_x = \{a_4, c_4\}$. Weights are $W(a_4) = 5$, $W(b_4) = 6$, $W(c_4) = 4$ and $W(d_4) = 3$. The middle column in Figure 9 shows an optimal solution to weighted coloring: $C_w(a_4) = [0, 4]$, $C_w(b_4) = [5, 10]$, $C_w(c_4) = [11, 14]$ and $C_w(d_4) = [5, 7]$. Although b_4 and d_4 are assigned to overlapping intervals, the solution is valid, for these two nodes do not intercept, i.e., they are in different cliques.

Weighted coloring is an NP-complete problem in most classes of graphs, including Interval Graphs. However, once

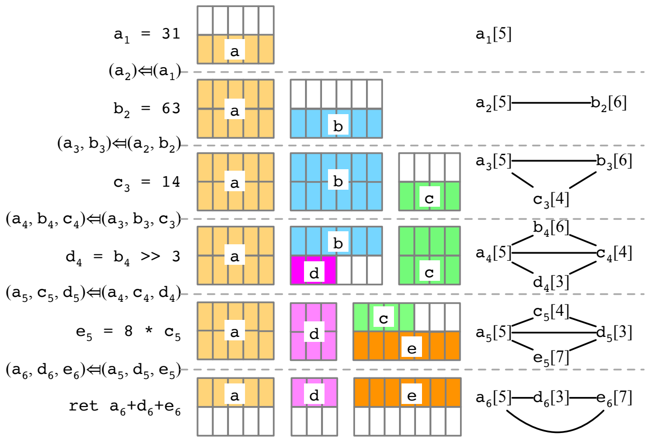

a_1 = 31
$(a_2) \Leftarrow (a_1)$
b_2 = 63
$(a_3, b_3) \Leftarrow (a_2, b_2)$
c_3 = 14
$(a_4, b_4, c_4) \Leftarrow (a_3, b_3, c_3)$
d_4 = b_4 >> 3
$(a_5, c_5, d_5) \Leftarrow (a_4, c_4, d_4)$
e_5 = 8 * c_5
$(a_6, d_6, e_6) \Leftarrow (a_5, d_5, e_5)$
ret a_6+d_6+e_6
a
b
c
d
e
$a_1[5]$
$a_2[5]$ $b_2[6]$
$a_3[5]$ $b_3[6]$ $c_3[4]$
$b_4[6]$ $a_4[5]$ $c_4[4]$ $d_4[3]$
$c_5[4]$ $a_5[5]$ $d_5[3]$ $e_5[7]$
$a_6[5]$ $d_6[3]$ $e_6[7]$

Figure 9. Solution to Weighted Graph Coloring problem for the elementary program seen in Example 3.1.

restricted to Elementary Graphs, the problem has a polynomial-time solution. In this case, WGCE is equivalent to finding the *Minimum Sum of Absolute Differences between two Lists* (MSUML)[2]. Given two lists—not necessarily of the same length—the problem asks for a matching between elements—not necessarily consecutive—that minimizes the absolute difference between paired elements. Figure 10 shows a solution to this problem, based on dynamic programming. Example 3.4 discusses a concrete instance of this problem.

Example 3.4. Consider two lists: Q1 = $[3, 8, 17]$ and Q2 = $[1, 3, 4, 7, 11, 16, 20]$. The matching that results in the minimum absolute difference is $(\mathtt{Q1}_0, \mathtt{Q2}_1)$, $(\mathtt{Q1}_1, \mathtt{Q2}_3)$ and $(\mathtt{Q1}_2, \mathtt{Q2}_5)$, assuming that indexation starts at zero. The absolute difference, in this case, is $|3 - 3| + |8 - 7| + |17 - 16| = 2$.

Theorem 3.5. *Figure 10 computes the minimum sum of absolute differences between arrays* Q1 *and* Q2.

Proof: The problem of minimum sum of absolute differences has an optimal substructure. Assuming that the length of Q1 is no smaller than the length of Q2, then we pair Q1[si] and Q2[bi] if the minimum sum of differences, up to that point is less than the cost of pairing Q1[si] with some element after Q2[bi]. The former alternative is Line 19 of Figure 10; the latter is Line 20. □

Reducing WGCE to the sum of absolute differences is trivial: given an elementary graph $G = (Q_a, Q_b, Q_x, W)$ we create two lists. The first, Q1 is formed by $\{W(e) \mid e \in Q_a\}$, sorted in ascending order. The other, Q2, is formed similarly, albeit with elements from Q_b. Assigning colors to elements in Q_x poses no problem: they can be colored within the interval $[0, w]$, where $w = \mathtt{sum}(W(e) \mid e \in Q_x)$.

[2]The only reference for MSUML is the "Dancing Cows" question in the *Sphere Online Judge* (at https://www.spoj.com/problems/DCOWS/).

```
1  Input:
2    int Q1[] # Assume w.l.g. that len(Q2)≥len(Q1)
3    int Q2[] # and that Q1 and Q2 are sorted.
4  Data structures:
5    int pair[len(Q1)]
6    int cache[len(Q2)+1][len(Q1)+1]
7  Output:
8    dp(0, 0)
9  where:
10   cache[len(Q2)][len(Q1)] = 0
11   for (int i = 0; i < len(Q2); i++) do
12   | cache[i][len(Q1)] = 0
13   for (int j = 0; j < len(Q1); j++) do
14   | cache[len(Q2)][j] = ∞
15 where:
16   int dp(int bi, int si) is
17   | if (cache[bi][si] >= 0)
18   | | return cache[bi][si]
19   | int t1 = |Q1[si] - Q2[bi]| + dp(bi+1, si+1);
20   | int t2 = dp(bi+1, si);
21   | if (t1 < t2) then
22   | | cache[bi][si] = t1;
23   | | pair[si] = bi;
24   | else
25   | | cache[bi][si] = t2;
26   | return cache[bi][si];
```

Figure 10. Solve WGCE by computing MSUML, the minimum sum of absolute differences between Q1 and Q2.

3.3 Elimination of the Elementary Format

We can solve the weighted coloring problem independently for each elementary graph that emerges from a program in elementary form. Figure 9 shows six solutions to this problem. However, to solve the Local Register Binding problem, each solution of WGCE must be merged into a single, cohesive assignment of registers to variables. Each solution of WGCE can be thought of as a bit-aware register assignment. For instance, if $C_w(v) = [l, u]$, then variable v will require a register with at least $u - l$ bits. To fuse successive solutions of WGCE, we resort to another matching problem: the *Minimum Sum of Absolute Differences of Same-Length Lists* (MSUMLL).

The only difference between MSUMLL and MSUML, the problem that Figure 10 solves, is the length of the lists. In the latter, this length might vary, whereas in the former, it should be the same. Therefore, Figure 10 can be used to solve also MSUMLL. However, this problem admits a much simpler solution, which Theorem 3.6 introduces and proves correct.

Theorem 3.6. *To compute the minimum sum of differences between two lists of equal length,* Q1 *and* Q2*: sort them in ascending order and pair up elements with the same index.*

Proof: Let $S = \{(s_1, u_1), \ldots, (s_n, u_n)\}$ be a solution to MSUMLL, where $s_i \in$ Q1 and $u_i \in$ Q2. Given two pairs, e.g., (s_x, u_x) and (s_y, u_y), if $s_x \leq s_y$ and $u_x \geq u_y$, by exchanging them, the resulting solution S' either has smaller sum

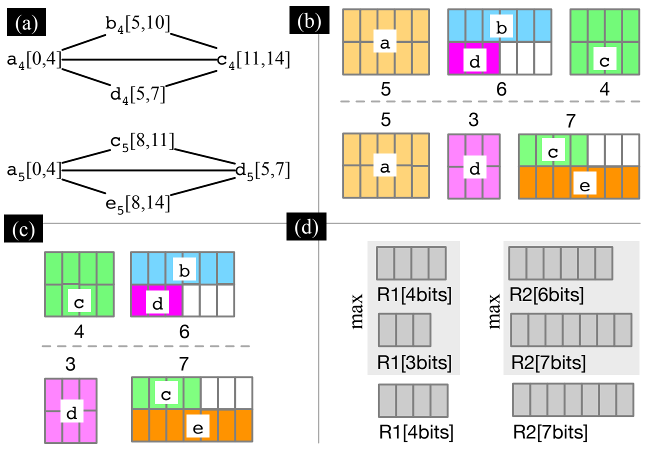


Figure 11. (a) Two solutions of WCEG to be merged. (b) Corresponding interval assignments. (c) The instance of MSUMLL that emerges from assignments. (d) A solution to MSUMLL. The maximum of each pair gives the register width.

(what would be an absurd) or have the same sum (and thus it is still an optimal solution). □

Updating the register bank. To merge two consecutive colored elementary graphs we proceed in three steps. We shall use Figure 11 to illustrate these actions. First, we remove variables that appear in the Q_x clique in both graphs. In Figure 11(a), such is the case of variable a_4. Removing such variable v does not compromise optimality, because v can be assigned to the same register in the two program points using exactly $C_w(v)$ bits. Then, we use the algorithm implied by Theorem 3.6 to solve the instance of MSUMLL that emerges from the two solutions of WCEG. Continuing with our example, Figure 11(b) shows the two instances of WCEG. Figure 11(c), in turn, shows the solution to MSUMLL, after removing variable a_4 from the problem. Finally, we take the maximum of each pair as the width, in bits, of the registers synthesized for these two program points. In Figure 11, we can allocate b_4, c_4, d_4 and e_4 with two registers, R1 with four bits, and R2 with seven bits. A third register, having five bits, will be assigned to a_4, given that $C_w(a_4)$ spans five colors. A solution to the LRB Problem guides the mapping of variables in elementary format to physical registers. Example 3.7 illustrates this mapping.

Example 3.7. Figure 12 shows the allocation produced for Example 2.2. The hardware synthesized for this example uses three registers, which add up to 16 bits. A swap between registers R1 and R2 is necessary to exchange variables c and d. The insertion of swaps follows known practice [27].

3.4 Complexity

The algorithm in Figure 10 is $O(|Q1| \times |Q2|)$. The size of Q1 or Q2 is the number of variables alive across consecutive

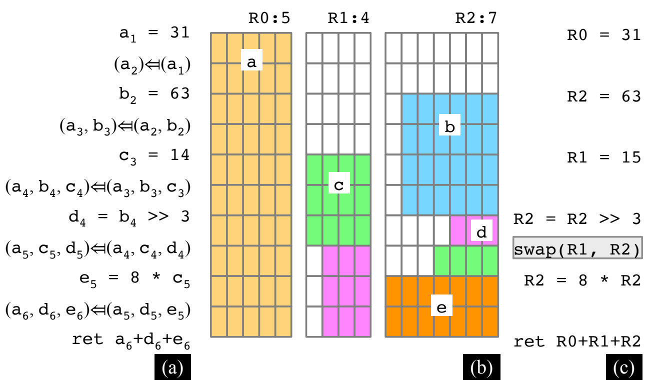


Figure 12. (a) Example 2.2 in elementary format. (b) Solution to LRB. (c) Instructions after register mapping.

instructions. If K is the maximum number of variables simultaneously alive; then Figure 10 is $O(K^2)$. Theorem 3.6 requires sorting these variables according to their bit widths. This step is $O(K \times \ln(K))$; hence, it is dominated by the complexity of Figure 10. Global register binding invokes Figure 10 $N - 1$ times, where N is the number of instructions on the program. Therefore, the register binding runs in $O(N \times K^2)$. In the worst case, $K \approx N$; hence, our approach is cubic on the number of program variables. In practice, K is much smaller than N; thus, we expect to observe linear behavior empirically. Indeed, Figure 13 (Page 8) indicates that although our algorithm is slower than the linear-time heuristics discussed in Section 2, the asymptotic behaviors of all these implementations follow similar patterns.

4 Evaluation

This section evaluates the following research questions:

RQ1 What is the time taken to do register binding?
RQ2 How does the optimal algorithm perform, in terms of number and size of registers?
RQ3 How many swaps must be inserted to ensure optimal register binding?

The Competing Approaches. This section compares the algorithm proposed in this paper (hencerforth called Swap) with the two heuristics discussed in Section 2 (Brisk and Cong). Additionally, we also include the original algorithm of Brisk et al. [2], which is not bit-aware. This algorithm, named 64Bk, assumes that all the registers have 64 bits.
Software: Algorithms run in LLVM 11.0.0. LLVM does not have a built-in range analysis. Thus, we have adopted the implementation of Quintão Pereira et al. [31]. As benchmarks, we use the 15 programs from the MIBENCH collection [15] that are publicly available in the LLVM test suite.
Hardware: Experiment were performed on an AMD Ryzen 7 1700 Eight-Core Processor with a clock of 3.0 GHz, and 64 GB of RAM (DDR4) operating at 2400 MT/s. The operating system used was Ubuntu Linux LTS version 20.04.

Linearization: The algorithms in this paper need to be incorporated into a global register allocator, because they are evaluated onto programs with branches. To meet this requirement, we have used them to support Poletto and Sarkar [29]'s linear scan register allocator, with the algorithmic extensions proposed by Sarkar and Barik [32]. The algorithm works per function: the basic blocks in a function are ordered, following a post-order traversal of its dominance tree. Once ordered, blocks are merged, thus yielding a so-called "super-block". The register allocators in this paper are then evaluated onto said super-blocks. Registers are joined across jumps through the so-called "parallel copies" (copies implemented with move and swap instructions). For a fully detailed description of this sequence of steps, we refer the reader to Sarkar and Barik's Figure 4 (Item 6). This technique to transform a local register allocator into a global one preserves optimality of SSA-based register allocation, as long as swapping the contents of registers is allowed. This result follows from Bouchez et al. [1]'s work. Bouchez et al. have shown that SSA-based register allocators, including Sarkar and Barik's, can decide in polynomial time if k registers are enough to compile a program whose largest number of variables alive at any program point is known. This result is true as long as register swaps are available.

Methodology: Register binding happens per function; hence, when reporting numbers per benchmarks, we are reporting totals accumulated *per function*. We have analyzed 492 functions from 15 different programs. Our implementation generates code into the so called "LLVM Machine-Code Intermediate Representation"; using, to this end, the infrastructure for register allocation already in place within LLVM.

Summary of Results. Table 1 summaries the results yet to be presented in this section. We shall be referring to this figure in the ensuing discussions.

Table 1. Summary of results. Running time given in milliseconds and register size given in bits. "Geo" is geometric mean, and "Avg" is arithmetic mean.

			Benchmark		Function	
		Total	Geo	Avg	Geo	Avg
Swap	Time	2048.3	1.91	21.21	0.28	4.16
	Size	92964	493.61	916.50	147.76	188.95
	Swaps	12562	20.33	122	6.35	25.53
Cong	Time	252.43	0.51	2.35	0.11	0.51
	Size	98269	515.08	940.71	154.83	199.73
Brisk	Time	327.41	0.54	3.42	0.09	0.67
	Size	102761	533.43	1002.68	158.87	208.82
64Bk	Time	327.41	0.54	3.42	0.09	0.67
	Size	115200	630.55	1122.17	189.51	234.15

4.1 RQ1 – Compilation Time

Figure 13 compares the time taken by the four algorithms evaluated in this section. The X-axis represents functions; however, the same point does not represent the same function for the different algorithms. Rather, functions are sorted by the time that each algorithm takes to process them. Thus, the first tick might represent different functions: namely, the function processed in the shortest time by each algorithm.

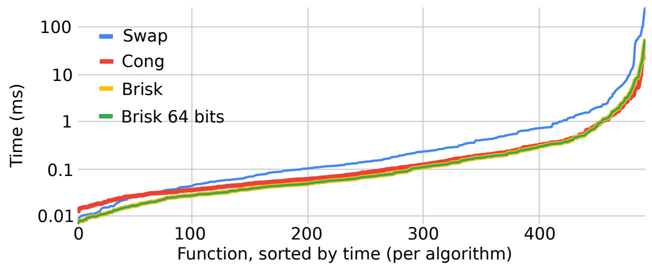


Figure 13. Time taken to solve register assignment. Each tick on the X-axis represents the i^{th} function processed by a particular algorithm. Functions are sorted, per algorithm, by their processing time. Thus, the area under each curve yields the total time taken by the corresponding algorithm.

Figure 14 highlights averages and outliers. Figure 15 shows time per benchmark. The time per benchmark is the sum of the time taken to perform register allocation on its constituent functions. The bottom part of Figure 15 shows how much slower we are compared to the other approaches.

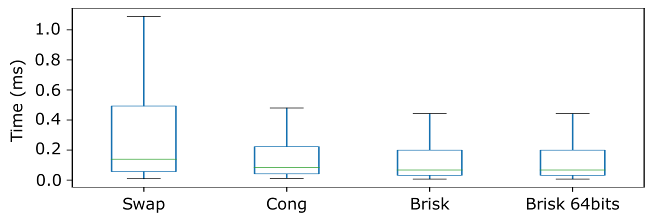


Figure 14. These box plots aggregate the time taken by each algorithm per each one of the 492 functions in LLVM's MiBench.

Discussion. Our approach is slower than the heuristics from Section 2. In total, it takes 2,048 seconds to process 492 functions. That is 8x longer than the time taken by Cong's approach, and 6x longer than the time taken by Brisk's algorithm. Yet, outliers influence these results heavily. If we consider averages (geometric mean per function), the difference is smaller: our approach is 2.5x slower than Cong et al.'s, and 3.1x slower than Brisk et al. If we group functions per benchmarks (Figure 15-Bottom), then, on average (geometric mean) our approach is 3.51x slower than Cong et al.'s, and 3.9x slower than Brisk et al.'s. Table 1 summarizes these totals. This result is expected: as seen in Section 3.4, we invoke a quadratic procedure for each pair of consecutive instructions in the linearized control flow graph. This step is absent in the heuristics of Brisk et al. and Cong et al.

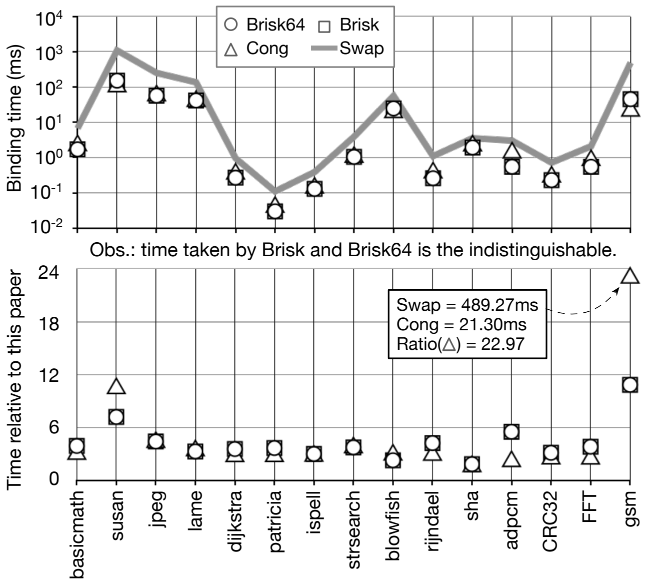


Figure 15. Time required to apply binding algorithms. Top: absolute time in milliseconds. Bottom: ratio using our algorithm as baseline (Number of times we are slower).

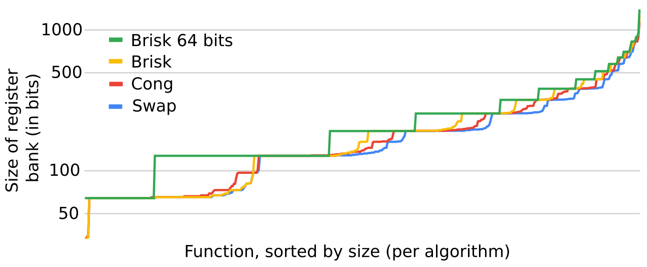


Figure 16. Size (in bits) of the register bank synthesized for each function in LLVM's version of MiBench. Each tick represents the i^{th} function processed by an algorithm. Functions are sorted by the size of the register bank that each algorithm has found for it.

4.2 RQ2 – Register Minimization

Figure 17 compares the different allocators in terms of bits necessary to implement register banks. We repeat the methodology of Figure 13 to show data: each tick on the X-axis represents one of 492 functions. Functions are sorted by the number of bits in the register bank found by each approach. Thus, the i^{th} tick might not correspond to the same function. Nevertheless, the area under each curve yields the total number of bits required by each allocator. The lower the area, the better the allocator. Figure 17 highlights averages and outliers. Finally, Figure 18 reports data per benchmark.

Discussion. The swap-based approach was the best, or tied as best, in every one of the 492 functions considered in this evaluation. This result was expected, as a consequence of that approach's optimality. The difference to the other allocators

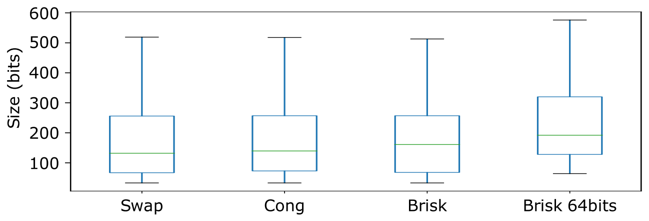


Figure 17. Box plots aggregating the size (in bits) of the register bank found by each algorithm for each one of the 492 functions in LLVM's MiBench.

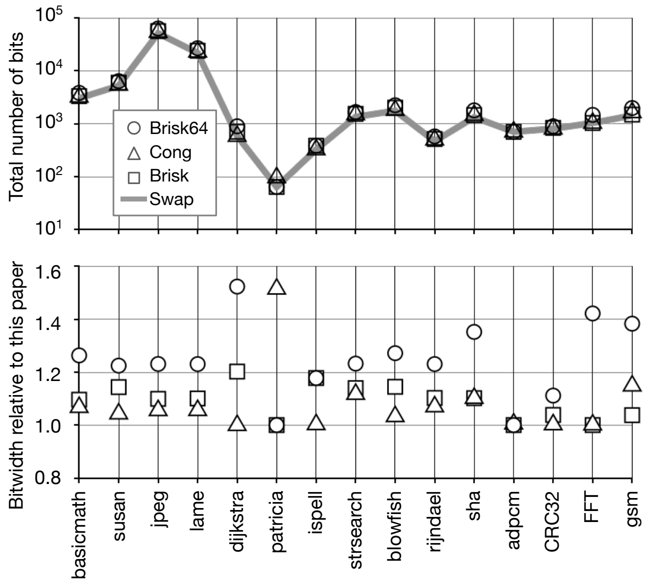


Figure 18. Number of bits (top: absolute; bottom: relative) used to compile the benchmarks from MiBench.

is small, albeit noticeable. Table 1 provides a summary of totals and averages. Our technique uses 6% fewer bits, in total, than Cong's, and 10% less than Brisk (with the bit-aware extension). If we use the original algorithm of Brisk et al., then the gap is larger: our register banks are 20% smaller.

On the Number of Registers. Our approach uses fewer registers (in addition to smaller ones). In total, it required 1,796 registers for all the 492 functions. Brisk et al.'s method (with or without the bit-aware extension) used 1,800. Cong et al.'s technique end up producing 1,982 registers. Thus, although Cong outperforms Brisk's method in the number of bits, it still requires more independent registers. This result is a natural consequence of Cong et al.'s objective function, which prioritizes bit width over the number of registers.

4.3 RQ3 – Number of Swaps

Our register allocator requires swapping values to achieve optimality. Figure 19 contrasts the number of swaps inserted in a function with the number of instructions in that function.

Each tick on the X-axis represents one of the 492 functions analyzed. Functions are sorted by the number of instructions that they contain. Figure 20, in turn, organizes this data per benchmark, juxtaposing the number of instructions with the number of swaps required per program.

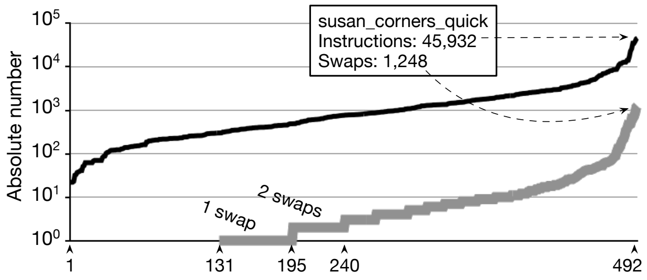


Figure 19. Comparison between the number of instructions and the number of swaps, considering 492 functions from MiBench. 363 functions had one or more swaps after register assignment. The X-axis represents functions, sorted by either the number of instructions or the number of swaps (There is no correspondence between ticks representing instructions and swaps). Numbers on the X axis represent how many functions had zero, at least one or at least two swaps.

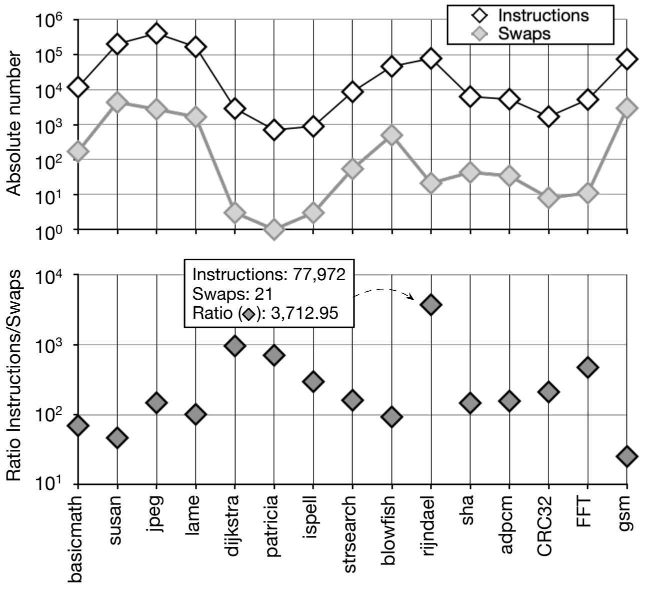


Figure 20. Number of instructions in program (not counting swaps) and the number of swaps inserted after bidding. Top: absolute values. Bottom: ratio of instructions per swaps.

Discussion. The 492 functions that constitute MiBench contain 1,024,805 LLVM instructions. Register binding required 12,561 swaps. Thus, one swap per 81 instructions. Swaps were required in 363 functions. There is a significant linear correlation between the number of instructions in a function and the number of swaps necessary to perform register binding on it. A regression model (excluding functions without swaps) gives us Swaps $= 0.18 \times$ Instructions$_{fun} - 12$, with $R^2 = 0.65$. If we group the functions per C file, then we get: Swaps $= 0.18 \times$ Instructions$_{file} - 45$, with $R^2 = 0.77$. However, most functions can be allocated without swaps, so if we consider the relation *per benchmark*, then we obtain: Swaps $= 0.009 \times$ Instructions$_{bench} + 64$, with $R^2 = 0.66$.

5 Related Work

The intersection between research in programming languages and high-level synthesis has created a field of study known as *silicon compilation*. Incidentally, inventions and discoveries made by groups more connected to one side of this intersection ended up being very important to the other.

Register Allocation. Optimal solutions for register assignment in SSA-form programs have been proposed in 2005-2006 [1, 2, 16, 24]. Among these techniques, only Brisk et al.'s targeted high-level synthesis. The other works were meant for the backends of general compilers. Those algorithms find optimal register assignments assuming uniform register banks, where all the registers have the same size and do not share bits. The algorithm of Cong et al. [7] was discovered independently from Brisk's, although at about the same time, and in the same institution. Cong et al.'s technique is not optimal; however, contrary to the works previously mentioned, it dealt with registers of different sizes. Its influence can be found in the design of algorithms for resource allocation and binding [6, 21] that are bit-aware.

Bit-width Analysis. Bit-aware resource binding requires a static analysis to infer the bit width of variables. If not the first, perhaps the most influential work along this line was the technique proposed by Stephenson *et al.* [34]. Stephenson's work is based on earlier ideas concerning the ranges of integer variables—themselves the origin of a field of research known as abstract interpretation [8]. Stephenson's techniques inspired a long string of contributions in the domain of high-level synthesis [14, 21, 30]. To evaluate our ideas, we have used an implementation of range analysis that was inspired by Stephenson's work [3, 31]. Nevertheless, any other implementation would suffice to our needs. Notice that bit-width analysis is also a fundamental component of solutions to the so called *register packing* problem. These solutions try to store multiple variables in different bits of the same register [10, 22, 36], assuming a fixed register bank. This problem differs from register binding in multiple ways. In particular, LRB's objective function (Def. 2.1) tries to minimize two goals: the number of registers drawn from an unlimited supply and their bit widths.

Allocation in High-Level Synthesis Tools. The two open-source HLS tools that we are familiar with, `LegUp` [4] and `Bambu` [28], use a heuristic to perform bit-aware resource allocation, following Stok [35]'s approach. Variables and storage units are grouped into bits according to the ranges inferred by some static analysis algorithm such as Gort and Anderson [14]'s (in LegUp) or Quintão Pereira et al. [31]'s (in Bambu). Variables with bit widths from 1 to 8, 9 to 16, 17 to 32 and 33 to 64 are considered compatible. To perform the allocation, each tool provides more than one heuristic—Huang et al. [17]'s bipartite weighted matching being the default choice (via its more modern interpretations [33]). These approaches are worse than Cong et al. [7]'s, because they tend to waist bits due to alignment constraints.

6 Conclusion

This paper has introduced an optimal solution to the bit-aware register binding problem. This algorithm finds, in polynomial time, the smallest register bank for any sequence of instructions. Contrary to the previous solutions to that problem, the new algorithm remains optimal in the presence of branches. In spite of these positive results, we do not expect this new algorithm to make prior heuristics obsolete. Our solution to register assignment, although polynomial-time, is still slower than preceding state-of-the-art techniques to solve the same problem. Furthermore, to ensure optimality, it inserts swaps in the program—action necessary to move variables among registers. Hence, minimizing swaps (an NP-complete problem) and speeding up the proposed technique are important future work. Nevertheless, it is our vision that the new algorithm sets a solid benchmark against which heuristics that synthesize register banks can be tried. In this regard, our experiments have shown that techniques such as Cong's or Brisk's algorithm still leave room for improvement, as they fall behind an optimal solution by 6 to 10%.

Acknowledgments

Fernando Pereira has been supported by CNPq (Grant 406377/2018-9); FAPEMIG (Grant PPM-00333-18) and CAPES (Edital CAPES PrInt). We thank Luigi Soares for reading a draft of this paper; and Fabrizio Ferrandi for kindly explaining the internals of Bambu to us. We also thank the anonymous referees and the anonymous shepherd allocated to this paper by Bernhard Egger, chair of the *ACM SIGPLAN 2022 International Conference on Compiler Construction.*

References

[1] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. 2006. Register Allocation: What does the NP-Completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How.. In *LCPC*. Springer, Heidelberg, Germany, 283–298. https://doi.org/10.1007/978-3-540-72521-3_21

[2] Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. 2006. Optimal register sharing for high-level synthesis of SSA form programs. *TCAD* 25, 5 (2006), 772–779. https://doi.org/10.1109/TCAD.2006.870409

[3] Victor Hugo Sperle Campos, Raphael Ernani Rodrigues, Igor Rafael de Assis Costa, and Fernando Magno Quintão Pereira. 2012. Speed and precision in range analysis. In *SBLP*. Springer, Brazil, 42–56. https://doi.org/10.1007/978-3-642-33182-4_5

[4] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, et al. 2013. From software to accelerators with LegUp high-level synthesis. In *CASES*. IEEE, USA, 1–9. https://doi.org/10.1109/CASES.2013.6662524

[5] Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6 (1981), 47–57. https://doi.org/10.1016/0096-0551(81)90048-5

[6] Deming Chen, Jason Cong, Yiping Fan, and Zhiru Zhang. 2007. High-level power estimation and low-power design space exploration for FPGAs. In *ASP-DAC*. IEEE, Singapore, 529–534. https://doi.org/10.1109/ASPDAC.2007.358040

[7] Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng. 2005. Bitwidth-aware scheduling and binding in high-level synthesis. In *ASP-DAC*. ACM, China, 856–861. https://doi.org/10.1109/ASPDAC.2005.1466476

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *POPL*. ACM, Austin Texas USA, 25–35. https://doi.org/10.1145/75277.75280

[10] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry V. Ponomarev. 2004. Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure. In *MICRO*. IEEE, New York, NY, USA, 304–315. https://doi.org/10.1109/MICRO.2004.29

[11] Bruno Escoffier, Jérôme Monnot, and Vangelis Th. Paschos. 2005. Weighted Coloring: Further Complexity and Approximability Results. In *ICTCS*. Springer, Heidelberg, Germany, 205–214. https://doi.org/10.1007/11560586_17

[12] Jonathan K. Lee et al. 2008. Aliased Register Allocation. *Theoretical Computer Science* 407, 1-3 (2008), 258–273. https://doi.org/10.1016/j.tcs.2008.05.025

[13] Martin Charles Golumbic. 2004. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., NLD.

[14] Marcel Gort and Jason H Anderson. 2013. Range and bitmask analysis for hardware optimization in high-level synthesis. In *ASP-DAC*. IEEE, Singapore, 773–779. https://doi.org/10.1109/ASPDAC.2013.6509694

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *WWC*. IEEE, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

[16] Sebastian Hack, Daniel Grund, and Gerhard Goos. 2006. Register allocation for programs in SSA-form. In *Compiler Construction*. Springer, Heidelberg, Germany, 247–262. https://doi.org/10.1007/11688839_20

[17] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu. 1991. Data Path Allocation Based on Bipartite Weighted Matching. In *DAC*. ACM, New York, NY, USA, 499–504. https://doi.org/10.1145/123186.123350

[18] Intel. 2020. *Intel High Level Synthesis Compiler Pro Edition*. Technical Report UG-20037/2020.06.22. Intel.

[19] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE, California, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[20] Jonathan K. Lee, Jens Palsberg, and Fernando Magno Quintão Pereira. 2007. Aliased Register Allocation for Straight-Line Programs Is NP-Complete. In *ICALP*, Vol. 4596. Springer, Heidelberg, Germany, 680–691. https://doi.org/10.1007/978-3-540-73420-8_59

[21] Ghizlane Lhairech-Lebreton, Philippe Coussy, Dominique Heller, and Eric Martin. 2010. Bitwidth-aware high-level synthesis for designing

low-power dsp applications. In *ICECS*. IEEE, Athens, Greece, 531–534. https://doi.org/10.1109/ICECS.2010.5724566

[22] V. Krishna Nandivada and Rajkishore Barik. 2013. Improved Bitwidth-Aware Variable Packing. *ACM Trans. Archit. Code Optim.* 10, 3, Article 16 (sep 2013), 22 pages. https://doi.org/10.1145/2509420.2509427

[23] Fernando Magno Quintão Pereira and Michael Canesche. 2022. *A Polynomial Time Exact Solution to the Bitwidth-Aware Register Binding Problem (Extended Version).* Technical Report LaC-2022-1. UFMG.

[24] Fernando Magno Quintão Pereira and Jens Palsberg. 2005. Register Allocation via the Coloring of Chordal Graphs. In *APLAS*. Springer, Heidelberg, Germany, 315–329. https://doi.org/10.1007/11575467_21

[25] Fernando Magno Quintão Pereira and Jens Palsberg. 2006. Register Allocation After Classical SSA Elimination is NP-Complete. In *FOSSACS*, Vol. 3921. Springer, New York, NY, USA, 79–93. https://doi.org/10.1007/11690634_6

[26] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register Allocation by Puzzle Solving. In *PLDI*. ACM, Tucson AZ USA, 216–226. https://doi.org/10.1145/1375581.1375609

[27] Fernando Magno Quintão Pereira and Jens Palsberg. 2009. SSA Elimination after Register Allocation. In *Compiler Construction*. Springer, Germany, 158–173. https://doi.org/10.1007/978-3-642-00722-4_12

[28] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *FPL*. IEEE, Washington, DC, USA, 1–4. https://doi.org/10.1109/FPL.2013.6645550

[29] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 895–913. https://doi.org/10.1145/330249.330250

[30] Suresh Purini, Vinamra Benara, Ziaul Choudhury, and Uday Bondhugula. 2020. Bitwidth customization in image processing pipelines using interval analysis and SMT solvers. In *Compiler Construction*. ACM, San Diego, CA, USA, 167–178. https://doi.org/10.1145/3377555.3377899

[31] Fernando Magno Quintão Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. 2013. A Fast and Low-Overhead Technique to Secure Programs against Integer Overflows. In *CGO*. IEEE, USA, 1–11. https://doi.org/10.1109/CGO.2013.6494996

[32] Vivek Sarkar and Rajkishore Barik. 2007. Extended Linear Scan: An Alternate Foundation for Global Register Allocation. In *Compiler Construction*. Springer, Heidelberg, Germany, 141–155. https://doi.org/10.1.1.94.4590

[33] Justus Schwartz, Angelika Steger, and Andreas Weißl. 2005. Fast algorithms for weighted bipartite matching. In *WEA*. Springer, Santorini Island, Greece, 476–487. https://doi.org/10.1007/11427186_41

[34] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. 2000. Bidwidth Analysis with Application to Silicon Compilation. *SIGPLAN Not.* 35, 5 (2000), 108–120. https://doi.org/10.1145/358438.349317

[35] Leon Stok. 1994. Data Path Synthesis. *Integr. VLSI J.* 18, 1 (1994), 1–71. https://doi.org/10.1016/0167-9260(94)90011-6

[36] Sriraman Tallam and Rajiv Gupta. 2003. Bitwidth Aware Global Register Allocation. In *POPL*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/604131.604139

[37] Donald Thomas and Philip Moorby. 2008. *The Verilog Hardware Description Language* (5th ed. ed.). Springer, Carnegie Mellon University (CMU).

[38] Xilinx. 2018. *Vivado Design Suite User Guide.* Technical Report UG902. Xilinx.