Improved Bitwidth-Aware Variable Packing

V. KRISHNA NANDIVADA, IIT Madras
RAJKISHORE BARIK, Intel Labs

Bitwidth-aware register allocation has caught the attention of researchers aiming to effectively reduce the number of variables spilled into memory. For general-purpose processors, this improves the execution time performance and reduces runtime memory requirements (which in turn helps in the compilation of programs targeted to systems with constrained memory). Additionally, bitwidth-aware register allocation has been effective in reducing power consumption in embedded processors. One of the key components of bitwidth-aware register allocation is the *variable packing* algorithm that packs multiple narrow-width variables into one physical register. Tallam and Gupta [2003] have proved that optimal variable packing is an NP-complete problem for arbitrary-width variables and have proposed an approximate solution.

In this article, we analyze the complexity of the variable packing problem and present three enhancements that improve the overall packing of variables. In particular, the improvements we describe are: (a) *Width Static Single Assignment* (W-SSA) form representation that splits the live range of a variable into several fixed-width live ranges (W-SSA variables); (b) PoTR *Representation* - use of powers-of-two representation for bitwidth information for W-SSA variables. Our empirical results have shown that the associated bit wastage resulting from the overapproximation of the widths of variables to the nearest next power of two is a small fraction compared to the total number of bits in use (≈13%). The main advantage of this representation is that it leads to optimal variable packing in polynomial time; (c) *Combined Packing and Coalescing* - we discuss the importance of coalescing (combining variables whose live ranges do not interfere) in the context of variable packing and present an iterative algorithm to perform coalescing and packing of W-SSA variables represented in PoTR. Our experimental results show up to 76.00% decrease in the number of variables compared to the number of variables in the input program in Single Static Assignment (SSA) form. This reduction in the number of variables led to a significant reduction in dynamic spilling, packing, and unpacking instructions.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*; *Compilers*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Variable packing, combined packing and coalescing

ACM Reference Format:
Nandivada, V. K. and Barik, R. 2013. Improved bitwidth-aware variable packing. ACM Trans. Architec. Code Optim. 10, 3, Article 16 (September 2013), 22 pages.
DOI: http://dx.doi.org/10.1145/2509420.2509427

1. INTRODUCTION

Applications from the embedded architecture domain (network, multimedia, and speech processing) extensively use narrow-width data via packing and unpacking.

This work is partially supported by the New Faculty Seed Grant, funded by IIT Madras CSE/11-12/567/NFSC/NANV.
Authors' addresses: V. K. Nandivada (corresponding author), Department of Computer Science and Engineering, IIT Madras, India; email: nvk@cse.iitm.ac.in; R. Barik, Intel Labs, Santa Clara, CA.

DOI: http://dx.doi.org/10.1145/2509420.2509427

A bitwidth-aware register allocation algorithm can potentially reduce the number of spilled variables by packing multiple of these narrow-width data items into a single physical register. The bitwidth-aware register allocator proposed by Tallam and Gupta [2003] consists of computing bitwidth information for variables at various program points using bit-section analysis and then packing narrower-width variables. This is followed by a traditional graph coloring register allocation. Tallam and Gupta show that optimal packing of variables is an NP-complete problem for arbitrary widths of variables (the problem can be seen as a variant of bin-packing problem) and present an approximate solution. In this article, we present new schemes to improve the bitwidth-aware variable packing and use the packing algorithm by Tallam and Gupta [2003] as a baseline for comparison.

One important fact in variable packing is that the live range of a variable may refer to different number of bits (we refer this as *useful* bits) at different program points. Note that, even though a variable is defined at only one place, the number useful bits might vary. Thus at every step of packing of variables, it is required to compute the useful bits for the newly generated packed variable (variable created after packing is done). The useful bits of the packed variable can be computed by either walking over the IR instructions again (time consuming) or *estimating* the useful bits using safe approximation. Tallam and Gupta [2003] computed Estimated Maximum Interference Width (EMIW) that approximates the actual Maximum Interference Width (MIW) of the packed variable for efficiency reasons. This step incurs bit wastage in favor of efficiency. To reduce such bit wastage (without losing efficiency), we introduce a new program representation called *Width Static Single Assignment* (W-SSA), which extends the classical Static Single Assignment (SSA) form [Cytron et al. 1991]. A program in our proposed W-SSA form guarantees that every variable is defined exactly once (similar to classical SSA), and additionally the set of useful bits of the variable remain unchanged throughout its lifetime. For instance, say a variable is defined only once, and has sixteen useful bits until a program point L1 and two useful bits thereafter then we will create two W-SSA variables: one of size sixteen bits (live until L1), and another of size two bits (live after L1). This gives a refined view of the bit usage, and helps reducing the bit wastage during the variable packing phase. In this article, we show that besides reducing the associated bit wastage, W-SSA form also aids in efficient packing and coalescing of variables.

Efficient representation of bitwidth information is an important requirement for different bitwidth-aware analysis including bitwidth-aware register allocation. Tallam and Gupta have used leading and trailing bit representation (LTR) for bitwidth information, wherein the width of a variable is represented by the start and end positions of the useful bits. We extend this further, and use the Powers-of-Two Representation (PoTR) for bitwidth information, wherein the sizes of the variables are always restricted to powers of two only. This requires overapproximation of the sizes to the nearest next power of two (for example, three variables that require 5, 7, and 8 bits are all assigned 8 bits each). Such a scheme can result in wastage of bits. Our experimental results show that the resulting wastage from such a scheme is a small fraction (around 13%) of the total number of bits required. The main advantage of PoTR is that it trivially leads to polynomial-time optimal packing of variables in bitwidth-aware register allocation [Coffman et al. 1987]. Note that, due to the incurred bit wastage, the resulting packing may not be truly optimal.

As can be easily seen, W-SSA form ensures uniform variable sizes, which makes for a simpler packing algorithm. However, W-SSA comes at a cost of creating more variables; while our packing algorithm does pack many variables, it still leaves some opportunities arising out of packing of noninterfering variables. We show that coalescing is an important aspect of variable packing, and propose a combined phase of variable

```
main() {
  ...
  w1 = C1 & 0xffff;
  w2 = C2 & 0xffff;
  w3 = C3 & 0xffff;
  w4 = C4 & 0xffff;
  L1 = ((w1 * K1) >> 16);
  L2 = ((w2 * K2) >> 16);
  L3 = ((w3 * K3) >> 16);
  L4 = ((w4 * K4) >> 16);

  print L1 + L2 + L3 + L4;
}
```
(a)

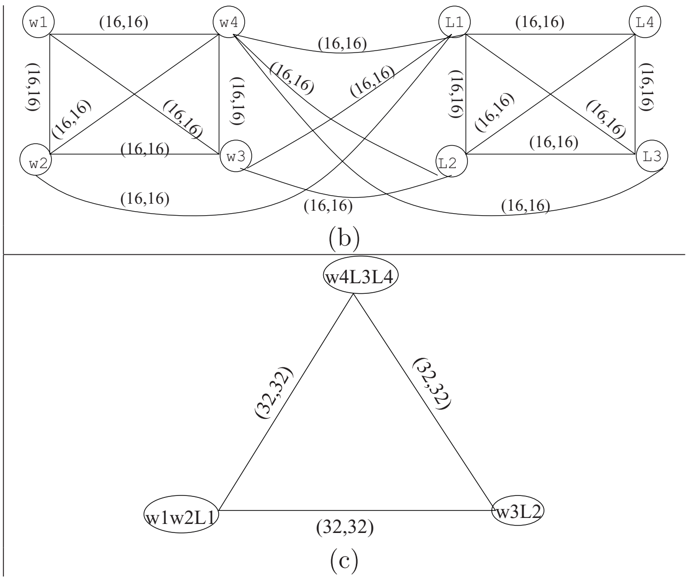

Fig. 1. (a) An example bitwidth-sensitive program (taken from the bilint benchmark of the BITWISE benchmark suite; (b) its interference graph. `C1, C2`..., and `K1, K2`... are constants and do not contribute to the nodes of the interference graph; (c) the resulting interference graph after applying the packing algorithm of Tallam and Gupta [2003].

coalescing and variable packing. We would like to recap the terms *packing* and *coalescing* of variables. Packing refers to combining two or more interfering variables so that the combined width is less than the width of a physical register [Tallam and Gupta 2003]. Coalescing refers to combining two or more noninterfering variables [Chaitin 1982]. Both packing and coalescing can be used to reduce register pressure[1] (and thus, improve the colorability of the interference graph) in a program. To illustrate the significance of coalescing on the packing process, we present a motivating example from the BITWISE benchmark suite.

Figure 1(a) shows a snippet from bilint benchmark (slightly modified to reduce complexity) from the BITWISE benchmark suite [Stephenson et al. 2000] and the interference graph annotated with the edge-weights [Tallam and Gupta 2003] is shown in Figure 1(b). An edge (n_1, n_2) has an associated edge-weight of (x_1, x_2), if the variable n_1 has at most x_1 number of useful bits while interfering with n_2, and n_2 has at most x_2 number of useful bits while interfering with n_1. In this particular interference graph, every variable has 16 useful bits each.

In this interference graph shown in Figure 1(b), the variable packing algorithm of Tallam and Gupta would pack at most two pairs of variables: Their technique packs two variables (merges two connected nodes in the graph and then updates the edge-weights of the affected edges), only if the resulting edge-weights are *valid*. An edge-weight (x_1, x_2) is considered *valid* if none of x_1 and x_2 is larger than the word size; for the sake of illustration, in this article we take the word size to be 32. Say we start by packing `w1` and `w2`, `L3` and `L4`, and `w3` and `L2`. We can further follow up by packing `w1w2` and `L1`, `w4` and `L3L4`. The final interference graph is shown in Figure 1(c). Further

[1]The number of colors needed to color an interference graph of a program is called the *register pressure* [Muchnick 1997] for the program. Register pressure at any program point p is the number of live variables at p.

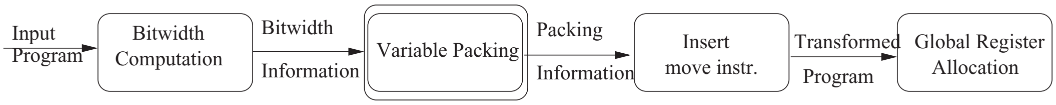

Fig. 2. Block diagram of the Tallam-Gupta bitwidth-aware register allocator.

packing of variables is not possible, as the resulting edge-weights would not remain valid (brief details about the Tallam-Gupta algorithm can be found in Section 2). This interference graph would require three registers. Let us now compare this result with the combined phase of variable coalescing and packing presented in this article: Our approach would first coalesce `w1` and `L1` into `w1L1`, `w2` and `L2` into `w2L2`, `w3` and `L3` into `w3L3`, and `w4` and `L4` into `w4L4`. Next, we pack `w1L1` and `w2L2` into a 32-bit variable, and `w3L3` and `w4L4` into another. We then invoke a traditional register allocator algorithm which would use only two registers during the register allocation phase.

This example demonstrates the importance of coalescing on variable packing.

The main contributions of this article include the following.

—We give an algorithm to transform a program in SSA form to W-SSA form.
—We provide the use of the powers-of-two representation for bitwidth information of W-SSA variables, and using the optimal algorithm of Coffman et al. [1987] for efficient packing, while incurring a modest bit wastage.
—We describe the problem of combined variable packing and coalescing. We identify it to be an NP-complete problem (reduces trivially to variable coalescing) and present a heuristic for combined variable packing and coalescing that further enhances the packing of variables.
—We give performance results to study the impact of variable packing and coalescing using BITWISE benchmark [Stephenson et al. 2000] suite. Our experimental results show decreases in the number of variables of up to 76.00% when compared to the original number of variables in the SSA form. This reduction in the number of variables led to a significant reduction in dynamic spilling, packing, and unpacking instructions (up to 100% reduction compared to that arising from the approach of Tallam and Gupta [2003]).

The article is organized as follows. In Section 2, we first present an overview of the Tallam and Gupta bitwidth-aware register allocator. In Section 3, we describe program representation using W-SSA form, PoTR representation, and an optimal variable packing algorithm that takes advantage of our representations. Section 4 presents our heuristic-based algorithm for the combined phase of variable packing and coalescing. We present our experimental results in Section 5. We discuss the related work in Section 6 and conclude in Section 7.

2. OVERVIEW OF THE TALLAM AND GUPTA REGISTER ALLOCATOR

In this section, we introduce the bitwidth-aware global register allocation algorithm presented by Tallam and Gupta [2003]. The overall block diagram and the algorithm for bitwidth-aware register allocation by Tallam and Gupta are given in Figure 2 and Figure 3 respectively. The key components of the algorithm are as follows.

—*Determining widths of variables:* (step 1) First, bitwidth of a variable is represented using the leading and trailing dead bits. Remaining middle bits are considered live and are not explicitly expressed. We term this representation as *LTR* width representation. Second, the authors propose a forward followed by a backward dataflow analysis to determine actual width of a variable at every program point. Both the dataflow analysis algorithms operate on a lattice over leading and trailing dead bit

```
1  Determine width of variables at various program points;
2  Build the Interference graph;
3  Associate labels to the edges of the interference graph based on the maximum
   interference width of two variables;
4  $PriorityList$ = Construct a prioritized list of all the variables;
5  while $\neg PriorityList.empty()$ do
6      $n = PriorityList.removeTop()$;
7      foreach node that $n$ interferes, say $n'$ do
8          if $n$ and $n'$ can be packed together into a single physical register then
9              Merge $n$ and $n'$ to a single node in the interference graph ;

10 Rename each packing variable and in the process introduce intra-register moves;
11 Rebuild the interference graph if needed;
12 Perform graph coloring based global register allocation;
```

Fig. 3. Bit-aware global register allocation.

pairs. Note that LTR ignores the fact that some of the bits in the middle live bit section may be dead.

—*Variable packing:* (steps 2–9) This is an iterative algorithm that at each step packs a pair of interfering variables into a single *packing* variable so that at no point their collective width is greater than the available number of bits in a physical register (given by the Maximum Interference Width (MIW) of the variables). Packing is performed on the interference graph whose edges are annotated with LTR width information. As nodes in the interference graph are packed, the LTR information for the packed variables are computed on-the-fly using Estimated Maximum Interference Graph (EMIW) for efficiency reasons. The packing algorithm tries to answer the following *key* question.
Key question. Given a set of variables and a constant *k*, does there exist a packing that reduces the number of variables to *k* such that the width of no packing variable is greater than the size of physical register?
This problem is shown to be NP-complete (by reducing it to a bin-packing problem), and the authors present a heuristic to prioritize the variables (and construct the *PriorityList*) to determine the order of packing.

—*Intra-variable moves:* (step 10–11) After packing is done, intra-variable move instructions are added to the IR for packing and extracting bits. This may require rebuilding the interference graph for the register allocation pass.

—*Register allocation* (step 12): A graph-coloring-based register allocator is invoked to perform allocation, coalescing, and assignment.

In this article, we present several enhancements to the variable packing phase and reuse the rest of the components of the traditional bitwidth-aware global register allocation algorithm.

3. IMPROVEMENTS IN VARIABLE PACKING

In this section, we propose two enhancements to variable packing. First, we describe a new Static Single Assignment (SSA)-based program representation that splits the live ranges of SSA variables into smaller fixed-width W-SSA variables based on their widths at various program points. The smaller live ranges of the fixed-width W-SSA variables create more opportunities for the packing algorithm. Second, we propose

L_1: $v = 2^{32} - 1$;
L_2: print v;

...
L_n: $x = v$ & 0xFFFF;
(a)

L_1 : $v_1 = 2^{32} - 1$;
L_2 : print v_1;
$v_2 = \Psi(v_1, 32, 0, 15)$
...
L_n : $x = v_2$ & 0xFFFF;
(b)

Fig. 4. A program and its corresponding W-SSA form.

Powers-of-Two Representation (PoTR) of width information for W-SSA variables. This leads to optimal packing at the cost of modest bit wastage.

3.1. Width-SSA (W-SSA) Form

A key property of a program represented in SSA form is that each variable is defined only once (and thus has a fixed value throughout its lifetime). This enables many powerful analyses such as partial redundancy elimination, sparse conditional constant propagation, and so on. Similar to the guarantee provided in an SSA program with respect to the value of a variable, it is desirable to have a constant actual width per variable throughout its lifetime. None of the current program representations that we are aware of gives such a guarantee.

Bitwidth-sensitive programs access different bit sections of a variable at different program points, and thus induce different sets of useful bits at various program points. We use this notion of useful bits to define the *actual width* of a variable.

Definition 3.1. Actual width is the set of contiguous bits of a variable that are actually required (*useful*) for its use or definition at a program point.[2]

We present an intermediate program representation called Width-SSA (W-SSA in short) which guarantees that each variable present in the program (called a W-SSA variable) has only one definition and has the same unchanged *actual width*. Figure 4 shows an SSA program fragment and its corresponding W-SSA translation. The code fragment in Figure 4(a) is in SSA form. Variable v requires 32 bits at L_1 and L_2. However, after L_2 the program uses only the lower 16 bits of v. That is, the actual width of v is not constant in this program (in SSA form). Figure 4(b) shows an equivalent program in W-SSA form. The original variable v has been split into two W-SSA variables v_1 (which requires 32 bits) and v_2 (which requires 16 bits). The narrow width of v after the print statement is captured using a select function Ψ that takes four arguments: (1) the source variable name; (2) the declared size of the source variable; (3) starting bit position; (4) the ending bit position. The function Ψ returns the required selection. The actual width of v_1 and v_2 are fixed throughout the program. That is, the actual width of v_1 is 32 bits and the actual width of v_2 is 16 bits[3] (the lower 16 bits of variable v).

Before we present the algorithm for W-SSA translation, we introduce some notations. We use the set `Vars` to denote the set of SSA variables in the program and `Nodes` to denote the set of nodes (statements) in the program. Note that, we do not treat the SSA ϕ nodes in any special way compared to the other nodes. We use a map `Use` : `Nodes` $\mapsto$ P(`Vars`); for any node n, `Use`(n) returns the set of variables used at n. Similarly, we use another map `Def` : `Nodes` $\mapsto$ P(`Vars`); for any node n, `Def`(n) returns

[2]We use *width* to refer to the starting and ending bits of the bit section of a variable as opposed to *size* which represents the number of bits in the width representation. Note that the actual width of variables at various program points can be computed using existing static bit-section analysis algorithms [Tallam and Gupta 2003; Stephenson et al. 2000; Barik and Sarkar 2006].

[3]It can be seen that the statement L_n in Figure 4(b) can now be copy-coalesced by the coalescing pass of register allocation with a copy instruction $x = v_2$.

```
1  function Rename-vars()
2  begin
3      wssaVars={};
4      foreach $n \in \texttt{Nodes}, v \in \texttt{Use}(n) \cup \texttt{Def}(n)$ do
5          replace the variable $v$ at node $n$, with $v_i$, where $i$ is $fresh$;
6          add $v_i$ to wssaVars;
7  end
8  function Build-W-dominators()
9  begin
10     $\texttt{WDom}(n_2, v_2) = \{(n_1, v_1)|$
       $n_1 \in \texttt{Nodes} \wedge v_1 \in \texttt{wssaVars} \wedge v_2 \in \texttt{Use}_w(n_2) \cup \texttt{Def}_w(n_2) \wedge v_1 \in \texttt{Use}_w(n_1) \wedge$
       $n_1 \in \texttt{Dom}(n_2) \wedge \Re(v_1, n_1) \subseteq \Re(v_2, n_2) \wedge$
       $(\neg \exists n_3 \in \texttt{Nodes}, v_3 \in \texttt{wssaVars} : (n_1, v_1) \in \texttt{WDom}(n_3, v_3) \wedge (n_3, v_3) \in \texttt{WDom}(n_2, v_2))\ \}$;
11 end
12 function Insert-$\Psi$-nodes()
13 begin
14     foreach $n_2 \in \texttt{Nodes}$ do
15         if $(n_1, v_1) \in \texttt{WDom}(n_2, v_2)$ then
16             Let $\Re(v_1, n_1) = (ll_1, ul_1)$;
17             Let $\Re(v_2, n_2) = (ll_2, ul_2)$;
18             insert $v_1 = \Psi(v_2, ul_2 - ll_2 + 1, ll_1, ul_1)$ after $n_2$;
19 end
20 function Eliminate-useless-vars()
21 begin
22     foreach $n_2 \in \texttt{Nodes}$ do
23         if $(n_2) \in \texttt{Dom}(n_1)$ then
24             Say two W-SSA variables $v_j$ and $v_i$ (with a common root variable) are used
               in $n_1$ and $n_2$ respectively;
25             if $\Re(v_j, n_1) \subseteq \Re(v_i, n_2)$ then
26                 Replace every occurrence of $v_j$ with $v_i$;
27                 Eliminate the $\Psi$ node that defines $v_j$;
28 end
```

Fig. 5. Transforming a program to W-SSA form.

the (singleton) set of variables defined at n in case of assignment statements, and an empty set otherwise. We also use the dominator map $\texttt{Dom} : \texttt{Nodes} \mapsto P(\texttt{Nodes})$; for any node n, $\texttt{Dom}(n)$ returns the set of nodes dominated by n.

In Figure 5, we present an algorithm to transform a given input program in SSA form to W-SSA form. The input SSA program is derived from an IR in three address code. The W-SSA translation algorithm consists of three phases.

(A) Rename-vars: This phase creates W-SSA variables by identifying each definition and use of SSA variables, and computes the set wssaVars. New W-SSA variables are created by breaking the live range of an SSA variable with varying actual widths. That is, the new live ranges of the W-SSA variables corresponding to an SSA variable and the union of the live ranges of the W-SSA variables matches the live range of the original SSA variable. In Figure 5, lines 3–5 replace each occurrence of an SSA variable v, with a new name (variable) v_i; v is called the root variable of v_i. We use different literals u, v, w to denote different SSA variables. We use the subscripted variables $v_1, v_2, \ldots, v_i$ to denote the W-SSA variables of v. We identify each def or use of a variable as a unique W-SSA variable.[4]

(B) Build-W-dominators: To represent the *w-dominator* (abbreviated for width-dominator) information, we define the map $\texttt{WDom} : (\texttt{Nodes} \times \texttt{wssaVars}) \mapsto P(\texttt{Nodes} \times \texttt{wssaVars})$. For example, $(n_1, v_1) \in \texttt{WDom}(n_2, v_2)$ indicates that W-SSA variable v_1 at node n_1 is width-dominated by W-SSA variable v_2 at node n_2. We say $(n_1, v_1) \in \texttt{WDom}(n_2, v_2)$ if v_2 can be used to extract the useful bits of v_1, n_2 strictly dominates n_1, and there exists no other (n_3, v_3) such that $(n_3, v_3) \in \texttt{WDom}(n_1, v_1)$ and $(n_2, v_2) \in \texttt{WDom}(n_3, v_3)$. A trivial yet correct approach to build WDom information could have been to treat the definition of the root SSA variable v as the w-dominator of all the W-SSA variables. This leads to longer live ranges for the W-SSA variables. We avoid this by identifying the closest w-dominator. Note that, the WDom map ensures that each W-SSA variable has at most one w-dominator and the W-SSA variables defined at any node have no w-dominators.

To build w-dominators, we use an utility map $\Re : (\texttt{wssaVars} \times \texttt{Nodes}) \mapsto (Int \times Int)$. For any variable v_i that is either defined or used at node n, $\Re(v_i, n)$ returns the lower and upper bit indices of the actual width of the variable v_i at node n.[5] The actual size of v_i at node n with $\Re(v_i, n) = (ll, ul)$ is $ul - ll + 1$. Given two range pairs $r_1 = (ll_1, ul_1)$ and $r_2 = (ll_2, ul_2)$, we say that $r_1 \subseteq r_2$ iff $(ll_2 \leq ll_1) \wedge (ul_1 \leq ul_2)$. Similar to the Use and Def maps, we define $\texttt{Use}_w$ and $\texttt{Def}_w$ maps to return the set of used and defined W-SSA variables, respectively.

In Figure 5, lines 7–11 compute WDom information. For the example program shown in Figure 4, $\texttt{WDom}(L_2, v_1) = \{(L_n, v_2)\}$.

(C) Insert-Ψ-nodes: To extract the relevant bits for a variable v_1 that is used at node n_1, we insert a Ψ node after n_2, where (n_2, v_2) w-dominates (n_1, v_1).[6] Note that we do not insert Ψ nodes at the Iterated Dominance Frontiers (IDF) [Muchnick 1997] unlike the insertion of ϕ nodes in SSA. This is done to avoid bit wastage that might result from carrying the extra bits from the IDF until the last use program point.

(D) Eliminate-useless-vars: The preceding presented steps may lead to creation of W-SSA variables that may have their live ranges and the actual widths completely contained within another. We invoke a cleanup optimization phase to replace all such variables with smaller range and width with the larger ones.

3.1.1. Complexity. The functions Rename-vars and Insert-Ψ-nodes have complexity linear in the number of nodes. In the case of Build-W-dominators, though the function could iterate over all the variables at each node, in reality, it needs to do so for only those variables that are used at that program point. For a program in 3-address code, this will always be a constant. Knowledge of this key point makes the complexity of the algorithm quadratic in the number of nodes ($O(n^2)$). The complexity of bit-section analysis (required for computing $\Re$) and dominators computation is $O(n)$ [Tallam and Gupta 2003; Alstrup et al. 1996]. Hence, the worst-case time complexity of the overall algorithm in Figure 5 is $O(n^2)$.

3.1.2. Code Generation. Similar to the code generation for SSA form, code generation for programs in W-SSA form would need to translate away the Ψ nodes. A typical (unoptimized) translation of a Ψ operation is given using a sequence of bitwise operations.

[4]This information can easily be refined using a global value numbering prepass. We leave this as future work.
[5]Since the input program is in SSA form, no node will both define and use the same variable. Also, since the input SSA program is derived from a program in three-address code, it will not have a case where a node has multiple uses of a variable with different actual sizes.
[6]If the source v_2 and destination v_1 both refer to the same set of bits of variable v, the select operator Ψ behaves like a copy operation. All these redundant copy instructions can be optimized away by a postpass of copy propagation after our algorithm.

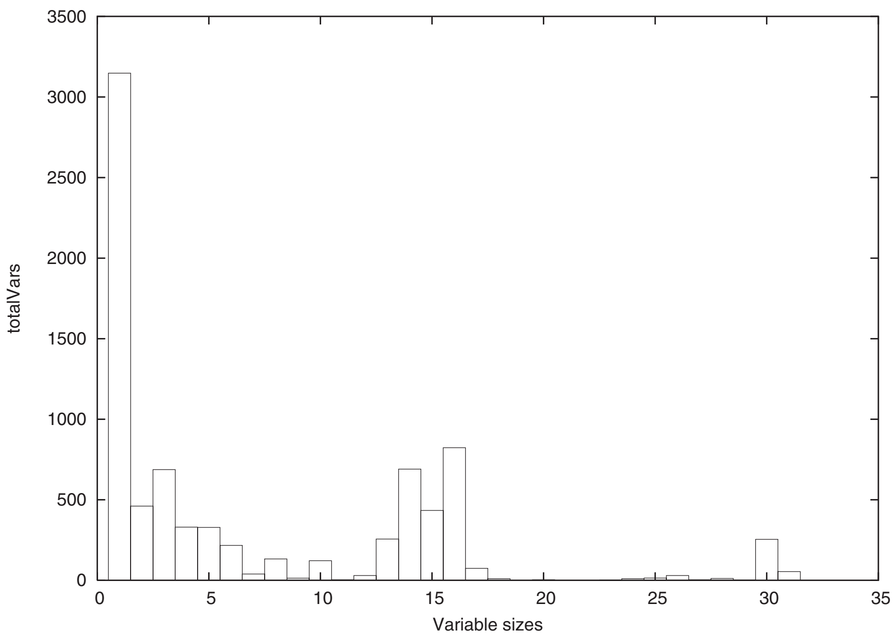

Fig. 6. Distribution of actual widths of variables in BITWISE benchmark suite.

$$v_2 = \Psi(v_1, s, ll, ul) \quad \Longrightarrow \quad \begin{array}{l} v_3 = v_1 \ll s - ul - 1 \\ v_2 = v_3 \gg ul + ll + 1 \end{array}$$

Here $\gg$ and $\ll$ represent the shift-right and shift-left operators respectively.

3.1.3. Correctness. We present an informal correctness argument. For the W-SSA transformation presented in Figure 5 to be correct, the following two points must hold.

—The W-SSA transformations preserve SSA form: *Intuition: Each* W-SSA *variable has exactly one w-dominator and thus is defined only once.*
—W-SSA transformation does not alter the semantics of the program: *Intuition: We do not introduce any new live ranges. We only subdivide existing live ranges into contiguous live ranges. Further, the union of the live ranges of the* W-SSA *variables matches the live range of the root SSA variable.*

3.2. Optimal Variable Packing Algorithm with PoTR

Tallam and Gupta have proved the NP-completeness nature of the variable packing algorithm. This is true where the actual width of a variable can have any value in the range 0 to the statically defined size of the variable. If we restrict the actual width of variables to the next-powers-of-two, then packing can be achieved optimally. In this section, we first present an empirical study of the bitwidth information in BITWISE benchmark set to demonstrate the modest bit wastage due to powers-of-two representation (PoTR) for the actual widths of W-SSA variables. Further, we discuss an optimal variable packing algorithm using the PoTR representation.

3.2.1. Bitwidth Representation. Figure 6 depicts the distribution of the actual sizes of variables in BITWISE benchmark suite [Stephenson et al. 2000] using a bit-section analysis described in Tallam and Gupta [2003]. BITWISE benchmark suite represents

```
1  function OptimalPacking($V$)
2  begin
3      Queue $Q$ = Sort $V$ in the decreasing order of the sizes of the W-SSA variables;
4      $Out$ = empty set;
5      $v'$ = create a new packing variable;
6      $used = 0$; Avail($v'$) = maxSize;
7      while $Q$.size() $\neq 0$ do
8          $v_i = Q$.remove();
9          packSetMap.add(($v_i$, ($v'$, $used$, $used$ + Size($v_i$) − 1))); // $v_i$ is packed inside $v'$
10         $used = used$ + Size($v_i$);
11         Avail($v'$) -= Size($v_i$);
           // Assert(Avail($v'$) $\geq 0$)
12         if $Avail\ (v') == 0$ then
13             $Out$.add($v'$);
14             $v'$ = create a new packing variable;
15             $used = 0$; Avail($v'$) = maxSize;
16     if $Avail\ (v') \neq Size(v')$ then $Out$.add ($v'$);
17     return $Out$;
18 end
```

Fig. 7. Optimal variable packing algorithm for W-SSA variables in PoTR.

a set of kernels from applications in the embedded systems domain. We calculate the total number of variables having different actual widths ($totalVars$) that are live at any program point across all the benchmark programs. We have plotted $totalVars$ against all possible subword variable sizes (1 bit – 31 bits, for 32-bit integers). One observation we make is that the distribution of actual sizes are clustered around numbers which are powers-of-two. This interesting observation is leveraged to represent the actual widths such that the actual size of each variable is a power-of-two irrespective of the statically defined size of the variable; we call it the powers-of-two representation (PoTR). Such a representation could result in the wastage of bits as each nonpowers-of-two-sized variable will be padded with extra bits to expand the size to the next number that is a power-of-two. Such a wastage could be significant. However, for the aforesaid benchmarks, we calculated the resulting bit wastage (computed as a percentage of extra bits required over the total number of bits) and found it to be around 13%. This modest bit wastage is tolerable since PoTR representation can help answer the *key* question presented in Section 2 in polynomial time (as shown in the reminder of this section) provided the size of the physical register is a power-of-two, which is the case in practice.

3.2.2. Packing Algorithm. Our optimal variable packing algorithm is based on the optimal bin-packing algorithm of Coffman et al. [1987]. Figure 7 presents the optimal variable packing algorithm. It takes a set V of W-SSA variables in PoTR and outputs a set Out containing the *packing* variables (the new packed variables). We assume that the map $Size$: `Vars` $\mapsto$ Int returns the size of the W-SSA variable, and $Avail$: `Vars` $\mapsto$ Int returns the number of available bits in the packing variable. The algorithm starts with a list of W-SSA variables sorted in the decreasing order[7]

[7]Irrespective of the order (increasing or decreasing) in which the W-SSA variables are sorted in line 3, the algorithm would still lead to optimal packing.

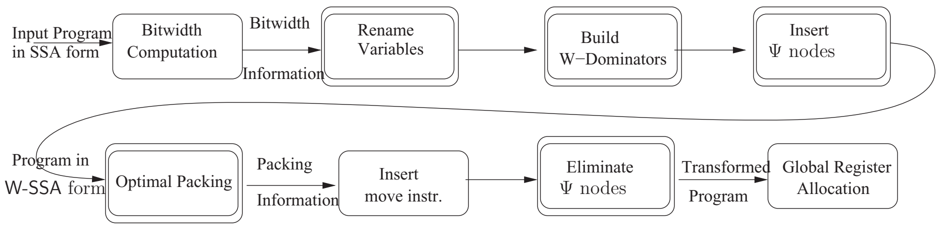

Fig. 8. Block diagram of the improved bitwidth-aware register allocator.

of their sizes and greedily packs them into the current packing variable. The set $\texttt{packMapSet} \subseteq \texttt{wssaVars} \times (\texttt{wssaVars} \times N \times N)$ is used to maintain the packing information; $(a, (b, i, j)) \in \texttt{packMapSet}$ indicates that the input W-SSA variable a is packed in the packing variable b from the bit positions i to j. The set *Out* is updated every time a packing variable gets full or when we break out of the while loop. The complexity of the algorithm is bound by the complexity of sorting ($O(n \log n)$).

3.2.3. Optimality of Packing Algorithm. Despite being greedy in nature, the algorithm presented in Figure 7 is optimal; the optimality result is derived from the optimality result of the bin-packing problem where the bin and the objects have powers-of-two sizes [Coffman et al. 1987].

Our packing result is optimal modulo the bit wastage resulting from PoTR representation of actual widths. Additionally, the optimal variable packing algorithm only solves the bin-packing problem without taking into consideration any possible constraints (interferences) between variables (see Irani and Leung [1996]).

3.3. Modifications to Bitwidth-Aware Register Allocation

Figure 8 shows the new block diagram for the bitwidth-aware register allocator as proposed in this section. The blocks with double lines are our contributions over Tallam and Gupta. Given bitwidth information from any bit-section analysis, we first translate the program into W-SSA form and round the actual widths of W-SSA variables to the next-powers-of-two. Subsequently, the variable packing algorithm takes as input the program in W-SSA form and generates the new *packing* variables. The packing variables and the W-SSA variables are eliminated in the Ψ elimination phase using additional intra-variable move instructions. In the end, we invoke the global register allocator (as done by Tallam and Gupta [2003]).

4. IMPROVING THE VARIABLE PACKING PRECISION BY COALESCING

Coalescing of variables was introduced by Chaitin [1982] and has been studied extensively in the context of register allocation. Coalescing of variables gets rid of some of the avoidable move instructions in the generated code. There exist several variants of coalescing: *aggressive* [Chaitin 1982], *conservative* [Briggs et al. 1994], *optimistic* [Park and Moon 2004], *iterative* [George and Appel 1996]. In the context of bitwidth-aware register allocation, as discussed in Section 1, the example program shown in Figure 1(a) demonstrates the significance of coalescing for better packing and in turn improved register allocation. In this section, we present a combined phase of variable packing and coalescing (cPAC). We first formulate the decision version of the cPAC problem and follow it up with a heuristic-based solution for the same.

Problem. Given an interference graph $G = (V, E)$ and two integers k_1 and k_2, does there exist a variable packing that can pack variables of V in k_1 number of 32-bit variables, such that at most k_2 noninterfering variables are not coalesced?

```
function SafelyAggressiveCoalesce($V$)
begin
    foreach $v_i \in V$ do $C(v_i) = \{$root variable of $v_i\}$;// Initialization
    Say the max size of any variable be $k = 2^r$;
    foreach $i = 0; i < r; i = i + 1$ do
        BucketCoalesce($i$); PackBucket($i$);
    BucketCoalesce($V, r$); CrossBucketCoalesce($V, 0, r - 1$); OptimalPacking($V$);
end
function BucketCoalesce($V, i$)// Coalesce in bucket $i$.
begin
    Worklist $w$ = SelectList( $V, i$); // all the $2^i$ sized variables from $V$
    CoalesceWorkList($w$);
end
function CoalesceWorkList($V, w$)
begin
    while $\neg w.empty()$ do
        $v_i = w.removeTop()$ ; // Removes one element
        if $\exists u_i \in w \wedge (u_i, v_i) \notin$ Interf then
            // Coalesce $u_i$, $v_i$.
            $w.remove(u_i)$; $V.remove(v_i)$; $V.remove(u_i)$;
            $x$ = create a new var; $w.add(x)$; $V.add(x)$;
            $C(x) = C(v_i) \cup C(u_i)$; UpdateInterf($v_i, x$); UpdateInterf($u_i, x$);
end
function PackBucket($V, i$)// Pack variables of size $2^i$
begin
    Worklist $w$ = SelectList( $V, i$);
    while $\neg w.empty()$ do
        $v_i = w.removeTop()$; // Removes one element.
        if $\exists u_i \in w \wedge C(v_i) \cap C(u_i) == \{\}$ then
            Assert$(v_i, u_i) \in$ Interf;
            $w$.remove($u_i$);
            $x$ = create a new var of size $2^{i+1}$ ; $V.add(x)$;
            $C(x) = C(v_i) \cup C(u_i)$ ; UpdateInterf($v_i, x$); UpdateInterf($u_i, x$);
end
function CrossBucketCoalesce($V, m, n$)
begin
    Worklist $w = emptyList$;
    foreach $i = m; i \leq n; i++$ do $w.add$(SelectList($V, i$));
    CoalesceWorkList($w$);
end
function UpdateInterf($v_i, x$)
begin
    foreach $(v_i, u_i) \in$ Interf do
        Interf = Interf $- \{(v_i, u_i), (u_i, v_i)\}$; Interf = Interf $\cup \{(x, u_i), (u_i, x)\}$;
end
```

Fig. 9. Combined packing and coalescing algorithm.

As a special case to the preceding problem statement, if the actual width of every variable is 32 bits, the problem trivially reduces to the graph coloring problem [Garey and Johnson 1979], which is NP-complete. Figure 9 presents a heuristic-based solution to the combined problem of coalescing and packing; it iteratively performs coalescing and follows it up with packing. Coalescing is performed aggressively like Barik and Sarkar [2006].

The function SafelyAggressiveCoalesce is the entry point of our algorithm. It takes as input a set V of W-SSA variables with PoTR representation of actual widths. The function first initializes a "contains" map C; C(v_i) returns the set of SSA root variables corresponding to all the W-SSA variables contained in v_i. Note that, because of packing and coalescing, a single *packing* variable may contain multiple W-SSA variables. For each possible *bucket* (denoted by the set of W-SSA variables having the same actual width), we iteratively coalesce all possible variables aggressively (BucketCoalesce) and then pack pairs of these coalesced variables into the next width bucket (PackBucket). Considering the complexity of coalescing algorithm [Bouchez et al. 2007], we avoid presenting any particular order among the variables ready for coalescing. In our implementation, the "SelectList" function returns a list of variables seen in the syntactic order of the program. At the end of the loop, BucketCoalesce is invoked to coalesce the variables of actual width 2^r that takes advantage of the noninterference among variables in the current bucket. At this point in the algorithm, there is a possibility that in each bucket zero or more W-SSA variables do not get packed and carried to another bucket (see the description of PackBucket). To coalesce and pack these variables (across buckets), we invoke the function CrossBucketCoalesce followed by OptimalPacking algorithm (Figure 7).

BucketCoalesce: In each bucket, we try to aggressively coalesce pairs of variables that do not interfere. We use a function SelectList : $\texttt{wssaVars} \times \mathit{Int} \mapsto P(\texttt{wssaVars})$; SelectList (V, i) returns a set (represented as a worklist) of variables from V, with actual width 2^i. The set Interf contains the set of all interfering pairs of variables. For each pair of noninterfering variables (v_i, u_i), we replace the pair with a coalesced variable that: (a) interferes with all the variables that v_i and u_i interfere with, and (b) contain all the variables that v_i and u_i contain. For the coalesced variable, line 21 updates the C map and the interference information (using the function UpdateInterf).

PackBucket: In this function, we try to pack pairs of variables of actual width 2^i into a variable of actual width 2^{i+1}. Note that, since we have already performed aggressive coalescing for the current bucket, every chosen pair of variables interfere. Hence, the assertion of line 29 in the code listing holds. We pack two variables only if their "contained" W-SSA variables do not share any common root variables. The reason is that, if two W-SSA variables v_1 and v_2, generated from a root variable v are packed into a single packing variable then we would be wasting more bits. After packing is completed, line 32 updates C map and the Interf set (using the function UpdateInterf).

4.1. Complexity

Each invocation of BucketCoalesce takes $O(|V|^2)$ time, where V is the set of W-SSA variables in the program. Note that Interf can be represented as a two-dimensional array and then the insertion, search, and delete operations all can be done in constant time. It may also be noted that function UpdateInterf takes $O(|V|)$ time. Function PackBucket takes $O(|V|^2)$ time. Since r is a constant in practice, the worst-case time complexity of our algorithm is $O(|V|^2)$.

4.2. Example

We now consider the example shown in Figure 1(b) and apply the algorithm presented in Figure 9. The transformation sequence, for one bucket (size 16) is shown in Figure 10. Note that unlike in Figure 1(b), we avoid showing the edge-weights (which is required for the approach of Tallam and Gupta [2003]) and instead show the size of the variables as an annotation on each node. In the fifth invocation of BucketCoalesce function (for $i = 4$), we coalesce w1 and L1 into w1L1, w2 and L2 into w2L2, w3 and L3 into w3L3, and w4

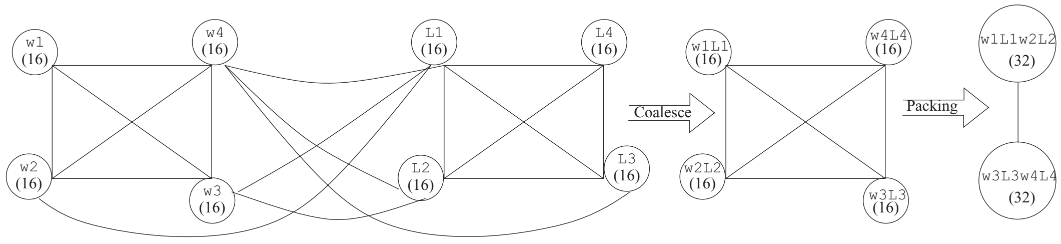

Fig. 10. Combined phase of coalescing and packing in action.

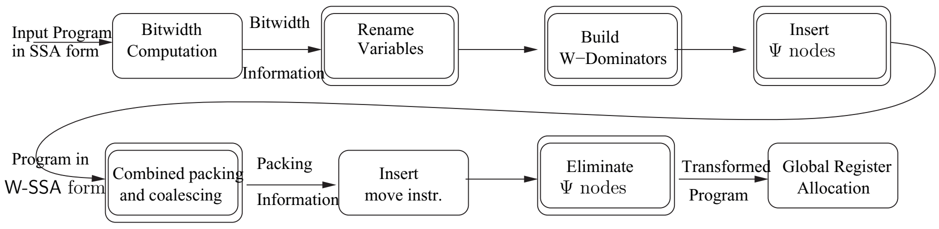

Fig. 11. Block diagram of the new bitwidth-aware register allocator using the combined packing and coalescing algorithm.

and L4 into w4L4. Next we invoke the PackBucket function and pack w1L1 and w2L2 into a 32-bit variable w1L1w2L2, and w3L3 and w4L4 into another 32-bit variable w3L3w4L4. This interference graph will require two physical registers during the register allocation pass.

4.3. Overall Bitwidth-Aware Register Allocation

Figure 11 depicts the final block diagram for the bitwidth-aware register allocator using the algorithm presented in this section. It is similar to the one presented in Section 3, except that the optimal packing algorithm has been replaced by a combined phase of coalescing and packing.

4.4. Safety in cPAC

While coalescing can lead to better packing of variables, arbitrary coalescing may produce a large number of packing variables after packing. We first present a safety criteria and argue that our combined coalescing and packing algorithm meets the safety criteria.

Definition 4.1. Safety criteria: A combined coalescing and packing algorithm (vCP) is safe with respect to a packing algorithm (vP) if for every input program P: $n_1 \leq n_2$, where n_1 is the number of packing variables generated by vCP for P, and n_2 is that generated by vP for P.

THEOREM 4.2. *The* SafelyAggressiveCoalesce *algorithm is safe with respect to the* OptimalPacking *algorithm shown in Figure 7.*

PROOF. (*Sketch*)
For an input program P, the number of packing variables that are produced after OptimalPacking is directly related to the sum of the total number of bits $\Sigma(V)$ of the variables present in V. This results in $\lceil \Sigma(V_s)/M \rceil$ number of packing variables, where M is the maximum possible width of a packing variable and V_s is the set of variables for a given size s.

For the same program P, we invoke SafelyAggressiveCoalesce(V_s). Let us assume that the set of variables considered by the OptimalPacking (invoked in line 7 of Figure 9) is given by V_e. It can be easily seen that $\Sigma(V_s) \geq \Sigma(V_e)$. That is, any invocation of the coalescing phase would never increase the total number of bits, because any successful invocation of the coalescing function (predicate in line 18 succeeds) would decrease it, and an unsuccessful coalescing operation would leave it unchanged. Similarly, the `PackBucket` function has no impact on the total number of bits. Thus, $\lceil \Sigma(V_s)/M \rceil \geq \lceil \Sigma(V_e)/M \rceil$, and hence, `SafelyAggressiveCoalesce` would not result in more number of packing variables than `OptimalPacking` when invoked on the same input program. Hence the proof. □

4.5. Discussion

Note that the safety criteria do not take into account the optimality of our cPAC algorithm in terms of the total number of coalesced variables and their actual widths. This is done deliberately to keep the issues of optimality and safety separate. We ensure that the absence of optimality does not lead to any amount of degradation in packing for the following phase.

Another point related to register allocation is that the packing phases of cPAC algorithm invoked via `PackBucket` do not affect the register pressure at any program point (only interfering variables of equal actual width are packed). However, the coalescing phases (invoked via `BucketCoalesce`, and `CrossBucketCoalesce`) and the global packing phase `OptimalPacking` may increase the register pressure. This can be easily overcome by modifying the coalescing and global packing algorithms with the help of a predicate that conservatively checks if the coalescing/packing under consideration increases the register pressure of the program and performs coalescing and packing only in cases where the register pressure is not increased. We use a two simple heuristics (that do not adversely affect the complexity of the algorithm) to address this issue.

—We only coalesce and pack non*simple* nodes. A node is simple if its degree is less than the total number of available physical registers. This way, we do not pack or coalesce variables which are guaranteed to get a register.
—We modify the `CoalesceWorkList` function, such that we coalesce variables with highest *dynamic total cost* first. We define dynamic total cost as the combined dynamic cost of storing variables to memory, loading variables from memory, and packing and unpacking of packed variables.

We use these heuristics in our empirical evaluation to establish the effectiveness in our introduced phases.

5. EXPERIMENTAL RESULTS

We now report on the results obtained from our prototype implementation of the optimal packing described in Section 3 and the combined packing and coalescing phase described in Section 4. The goal of this section is twofold: (a) to evaluate the PoTR and W-SSA representation; (b) to empirically establish the impact of coalescing on packing. The benchmarks were all taken from the BITWISE benchmark set [Stephenson et al. 2000]. All implementations were carried out in GCC 4.1 framework targeted to x86 platform. We have tested our analysis at -O2 level of optimization of gcc. In our implementation, we avoid rounding the widths and just use a greedy packing on a sorted list of variables. Note that this strategy will never do worse than the case where the widths are rounded off, and in some cases may do better. The changes to the (basic) algorithms in Figure 7 and Figure 9 are trivial and we only present differences from the basic versions.

—In Figure 7, the "Assert" after line 10 is eliminated.
—In Figure 7, line 11 is replaced with
"**if** Avail $(v') \leq 0$ **then**"

		TG			Our algorithms. Sec 3, Sec 4				Overall impr	
benchmark	# orig SSA vars	# vars after PKG	% less over orig	# W-SSA vars	#vars after opt OPK	% less over W-SSA	#vars after CPC +OPK	% less over W-SSA	% less over orig	% less over TG
adpcm	26	19	26.9	59	34	42.4	12	79.7	53.8	36.8
bilint	12	11	8.3	17	10	41.2	3	82.4	75.0	72.7
bubblesort	20	18	10.0	30	20	33.3	8	73.3	60.0	55.6
convolve	8	8	0.0	15	10	33.3	4	73.3	50.0	50.0
edge_detect	104	103	1.0	130	110	15.4	26	80.0	75.0	74.8
histogram	29	28	3.4	44	29	34.1	8	81.8	72.4	71.4
jacobi	36	32	11.1	60	50	16.7	11	81.7	69.4	65.6
levdurb	37	37	0.0	57	45	21.1	13	77.2	64.9	64.9
life	47	47	0.0	68	49	27.9	12	82.4	74.5	74.5
median	33	33	0.0	47	39	17.0	13	72.3	60.6	60.6
motiontest	12	12	0.0	16	13	18.8	9	43.8	25.0	25.0
mpegcorr	30	29	3.3	44	33	25.0	10	77.3	66.7	65.5
newlife	62	61	1.6	87	71	18.4	13	85.1	79.0	78.7
sha	610	610	0.0	618	616	0.3	30	95.1	95.1	95.1
softfloat	581	534	8.1	878	758	13.7	243	72.3	58.2	54.5

Fig. 12. Comparison of our algorithms presented in Section 3 and Section 4 vs. variable packing algorithm developed by Tallam and Gupta (TG) for optimized code (using -O2).

—In Figure 9, line 3 is replaced with
"Say the max size of any variable be $k \leq 2^r$; // for the smallest r."

Figure 12 compares the performance of packing and coalescing algorithms described in this article against the packing algorithm described in Tallam and Gupta [2003].[8] The bitwidth information for all the variables across all program points is computed using the static analysis described in Tallam and Gupta [2003] and is provided as an input to all the algorithms. We have abbreviated the Tallam and Gupta "packing phase" with PKG, "optimal packing phase" with OPK, and the "combined packing and coalescing phase" with CPC.

For our packing and coalescing algorithms, we present the input programs in W-SSA form. The number of W-SSA variables are shown in column 5 of Figure 12. Since multiple W-SSA variables are created for varying sizes of an SSA variable, the number of W-SSA variables generated (column 5) is either equal to or more than the number of original SSA variables (column 2). On average the number of W-SSA variables were found to be around 40% more than the number of SSA variables.

Comparing the number of packing variables obtained by using Tallam and Gupta [2003] (column 3) and the packing algorithm described in Section 3 (column 6), it can be seen that the performance of our packing algorithm is comparable. However, Tallam and Gupta is consistently performing better. Further investigations revealed another interesting issue in our packing algorithm: in `edge_detect` benchmark we found that one pseudo-variable had initial size of 16 bits, and subsequent size of 32. This resulted in the creation of two W-SSA variables and our algorithm ended up using two packing variables, whereas the Tallam Gupta approach uses a single 32-bit variable across all program points. This reinforces our original motivation for combined coalescing and packing algorithm to enhance bitwidth-aware register allocation algorithm performance.

The coalescing and packing algorithm described in Section 4 significantly improves the number of (W-SSA) packing variables. In the best case, it improves up to 95.1%

[8]We have avoided the compile-time numbers as we did not see any visible deterioration in compile time.

in column 10 of Figure 12 (for `life` benchmark). The geometric average improvement was found to be 62.99%. It can also be seen that our combined coalescing and packing algorithm significantly outperforms the variable packing obtained by Tallam and Gupta (up to 95.1%, geometric average 60.32%). We conclude from such an evaluation that coalescing has significant impact on the performance of packing algorithm in bitwidth-aware register allocation.

To establish the real benefit of packing, we define a metric *dynamic total cost* as the total cost of the dynamic spill loads, spill stores, packing, and unpacking; the execution frequencies are obtained by profiling the applications.[9] Now, we compare the performance of our proposed algorithm by studying the impact of our algorithm and that of Tallam and Gupta on dynamic total cost.

For each of the benchmarks discussed in Figure 12, we compare, in Figures 13 and 14, the impact on dynamic total cost in the presence of three packing algorithms[10]: (i) packing algorithm of Tallam and Gupta (TG - red line). Since we are comparing Tallam and Gupta with our packing algorithm that includes aggressive coalescing, we invoked a postpass of aggressive coalescing [Budimlic et al. 2002] and to be fair used the lower[11] of the following two dynamic total costs: Tallam-Gupta + aggressive coalescing, and stand-alone Tallam-Gupta; (ii) our combined phase of coalescing and packing (cPAC - green line); (iii) our combined phase of coalescing and packing supported by the heuristic discussed in Section 4.5 (cPACH - blue line). For varying number of available registers (6, 8, 10, 12, 16, 18), we plotted the dynamic total cost for each of the three methods. We can make the following observations.

—For four benchmarks TG performs better for fewer number of registers (adpcm and bubblesort for six registers, life up to ten registers, and motiontest to eight registers). For the first two of these benchmarks, cPAC and cPACH catch up and outperform TG in the presence of 8 or more registers. For the latter two benchmarks, cPAC and cPACH catch up with TG for higher number of available registers. The reason behind this is that our greedy heuristic mentioned in Section 4.5 does not currently take into account the packing and unpacking overheads explicitly (it primarily focuses on register pressure). Note that both cPAC and cPACH lead to much better spill cost compared to TG (and also the total number of coalesced and packed variables as shown in column 8 of Figure 12) for these benchmarks at lower registers, but the cost of packing and unpacking mitigates these spill cost benefits. This we believe can be easily fixed by extending our current heuristic to coalesce while also controling the packing and unpacking costs.
—For the convolve benchmark all the three algorithms incur in zero cost (and hence all the three curves are superimposed on top of one another).
—For blint, cPAC and TG incur the same cost and thus are superimposed. In jacobi, cPAC and cPACH are superimposed on each other as our greedy heuristic does not find enough opportunities to improve the code.
—Overall both cPAC and cPACH outperform TG for most of the observed points.
—cPACH performs better cPAC on nearly all of the benchmarks; this signifies the relevance of our heuristic.
—It can be seen that for each benchmark, all the three curves flatten out for increasing number of registers. The flattened portions correspond to the packing and unpacking

[9]For the benchmark “softfloat”, the base gcc compiler was giving a SEGFAULT while collecting the profile information. We had to disable the profiling for one function (estimateDiv64To32) to include the results for the benchmark; we instead used static frequencies for the loops in the function.
[10]In practice all the spills may not be equal and depending on the architecture the load and store costs may differ.
[11]Note: coalescing may increase the total dynamic cost under some circumstances.

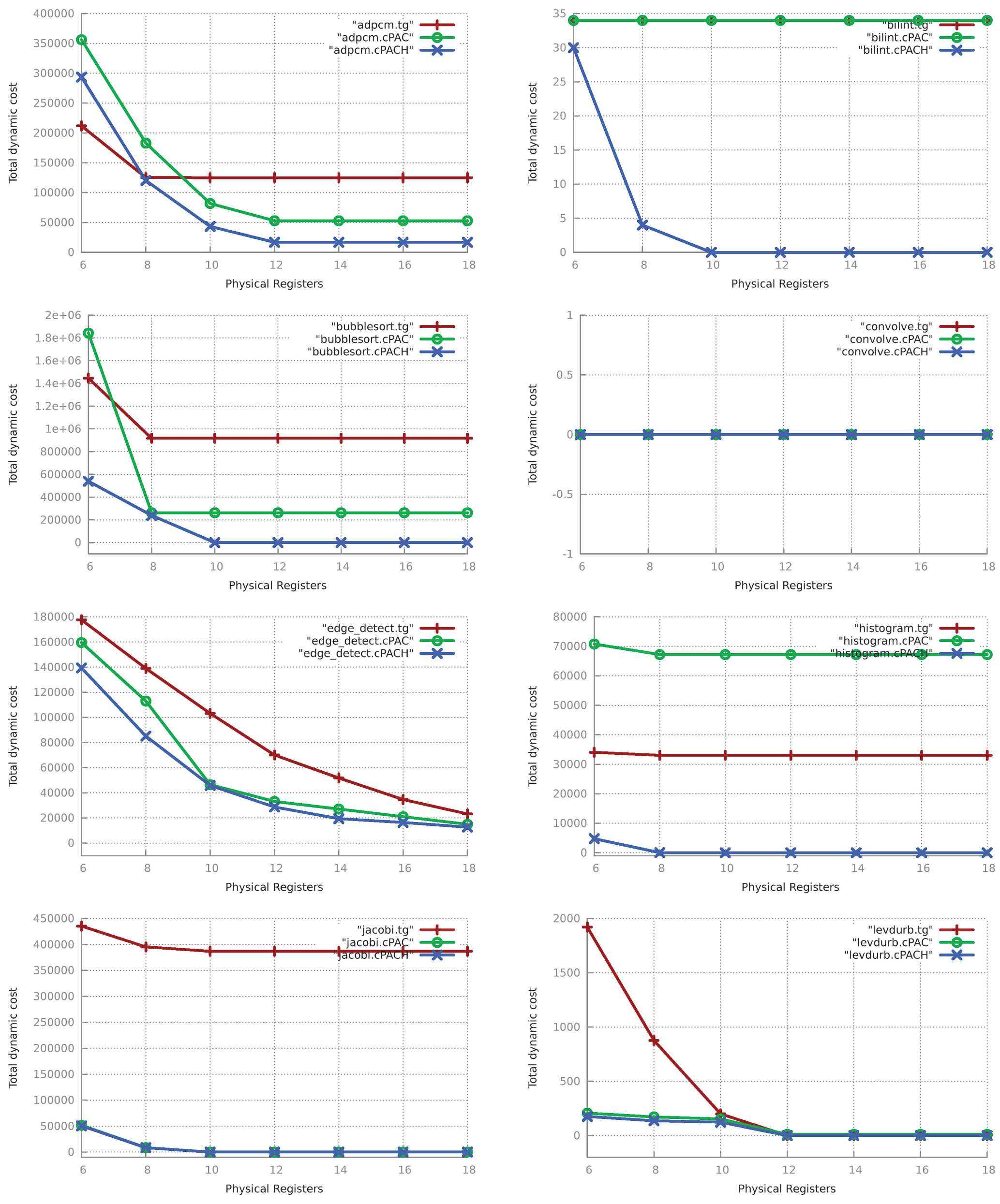

Fig. 13. Plot I: Comparison of dynamic total cost (= spill load cost + spill store cost + packing cost + unpacking cost). Plots captions from left to right: adpcm, bilint, bubblesort, convolve, edge_detecteps, histogram, jacobi, and levdurb. We assume the relative cost of a spill load or a store instruction is 4 compared to the cost of pack/unpack instruction is 1; this is consistent with the costs assumed by the underlying gcc framework for different analyses and transformations.

cost which is incurred independent of the number of registers; the spill load/store cost becomes zero with increasing number of registers.

—Owing to improved packing of variables we incur less spill cost and slightly higher packing/unpacking cost.

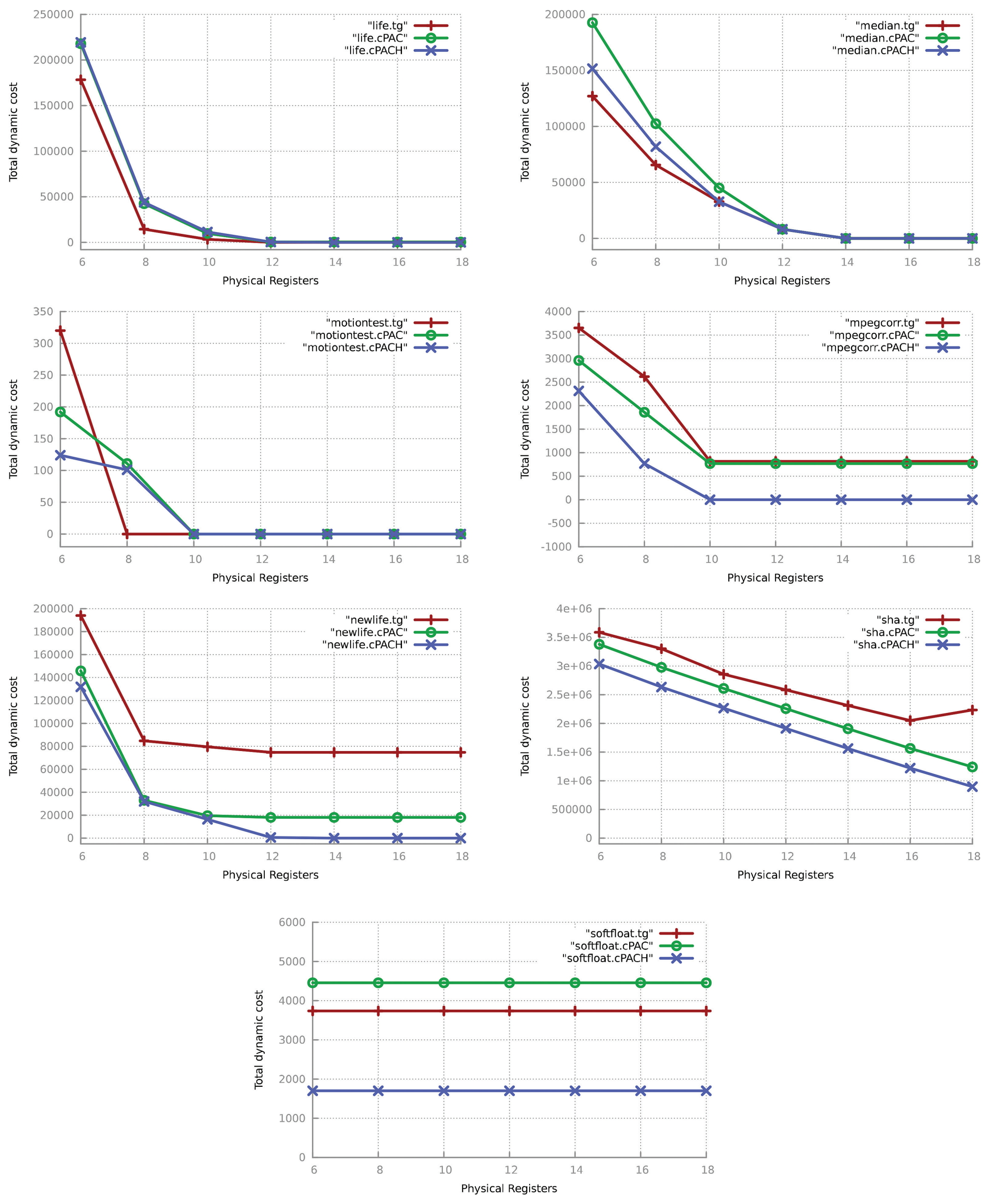

Fig. 14. Plot II: Comparison of dynamic total cost (= spill load cost + spill store cost + packing cost + unpacking cost). Plots captions from left to right: levdurb, life, median, motiontest, mpegcorr, newlife, sha.

6. RELATED WORK

Traditional SSA form [Cytron et al. 1991] keeps track of definition and uses of scalar variables. Several research activities have undergone in extending traditional SSA form to represent various attributes (Concurrent-SSA [Lee et al. 1997], Array-SSA [Knobe and Sarkar 1998]) in a particular domain. Our W-SSA form is another such representation that splits original program live ranges into smaller ones which have unique width throughout their lifetime. We have shown that such a representation aids in efficient variable packing.

Recently, narrow-width values have caught the attention of researchers, and they have explored both hardware- [Ergin et al. 2004; Lipasti et al. 2004; Kondo and Nakamura 2005], and compiler-assisted [Tallam and Gupta 2003] techniques to pack multiple values into a single physical register. In this article, we take inspiration from the compiler-assisted techniques of Tallam and Gupta [2003] and present new efficient measures to pack variables containing subword data in an efficient way.

Tallam and Gupta [2003] introduced the notion of bitwidth-aware register allocation in the compilers. They provide a heuristic to pack narrow-width data that uses an interference graph annotated with edge labels based on the maximal interference of variables. Interfering variables are packed iteratively until no more packing can be done. New edge labels are computed on-the-fly using heuristic estimates that obey the intermediate value theorem [Tallam and Gupta 2003]. A new set of estimates are proposed in Barik and Sarkar [2006] that improve the packing algorithm. Both Barik and Sarkar [2006] and Tallam and Gupta [2003] pack variables without coalescing them during packing, which our approach does.

Coffman et al. [1987] discuss the properties of powers-of-two in the bin-packing problem. Bar-Noy et al. [2004] discuss optimal solutions for windows scheduling problem with powers-of-two windows. In this article, we apply a similar solution to variable packing.

Coalescing is well-studied in the context of register allocation: conservative coalescing by Briggs et al. [1994], aggressive coalescing by Budimlic et al. [2002], optimistic coalescing by Park and Moon [2004], and iterative coalescing by Appel and George [2001]. Bouchez et al. [2007] present a good account of the complexity of different variations of the coalescing problem. To our knowledge, coalescing in the presence of narrow-width data has not been studied in the literature. Our article addresses this issue in the presence of W-SSA form and powers-of-two representation and provides a heuristic for a combined packing and coalescing phase.

7. CONCLUSION AND FUTURE WORK

In this article, we have presented improvements to the variable packing algorithm of the bitwidth-aware register allocation algorithm proposed by Tallam and Gupta [2003]. We propose modifications to both bitwidth representation and program representation via PoTR and W-SSA, respectively, to realize the improvements. We show that variable coalescing is an important ally of variable packing and present an iterative aggressive coalescing-based packing algorithm. Our experimental results show decreases in the number of variables of up to 76.00% when compared to the original number of variables in the SSA form. Our approach led to a significant reduction in dynamic spilling, packing, and unpacking instructions. We get these improvements at the cost of transforming the input program into W-SSA form and reverting it back. We are currently exploring how other optimizations such as memory coalescing may be able to take advantage of code in W-SSA form.

Rigorous performance evaluation of our generated optimized code on a cycle-accurate simulator that admits efficient packing/unpacking instructions would be one of the key challenges that we leave as future work. Another interesting future work is to extend our variable packing algorithm to efficiently pack in registers with varying sizes; this is important for architectures like Intel that allow efficient access of partial registers : different parts of the EAX can be accessed as AX (lowermost 16 bits), AL (lowermost 8 bits), or AH (bits 8–16). Another interesting future work is the possibility of directly incorporating our algorithm inside the register allocation pass instead of keeping them separate. Since both these components have a phase ordering issue between them, such a combined approach might deliver better results.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments and suggestions on the past submissions related to this article. In particular, the idea about the greedy approach to improve the actual packing (described in the beginning of Section 5), and the basic intuition behind the Eliminate-useless-vars were provided by the reviewers of ACM TACO.

REFERENCES

ALSTRUP, S., LAURIDSEN, P. W., AND THORUP, M. 1996. Dominators in linear time. DIKU Tech. rep. 35. University of Copenhagen.

APPEL, A. W. AND GEORGE, L. 2001. Optimal spilling for cisc machines with few registers. In *Proceedings of the Conference on Programming Language Design and Implementation (SIGPLAN'01)*. 243–253.

BAR-NOY, A., LADNER, R. E., AND TAMIR, T. 2004. Windows scheduling as a restricted version of bin packing. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*. 224–233.

BARIK, R. AND SARKAR, V. 2006. Enhanced bitwidth-aware register allocation. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06) held as part of the Joint European Conferences on Theory and Practice of Software.* 263–276.

BITWISE BENCHMARKS. 2013. http://www.cag.lcs.mit.edu/bitwise/bitwise_benchmarks.htm

BOUCHEZ, F., DARTE, A., AND RASTELLO, F. 2007. On the complexity of register coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization*. 102–114.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16,* 3, 428–455.

BUDIMLIC, Z., COOPER, K. D., HARVEY, T. J., KENNEDY, K., OBERG, T. S., AND REEVES, S. W. 2002. Fast copy coalescing and live-range identification. *SIGPLAN Not. 37,* 5, 25–32.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. *SIGPLAN Not. 17,* 6, 98–105.

COFFMAN, JR., E. G., GAREY, M. R., AND JOHNSON, D. S. 1987. Bin packing with divisible item sizes. *J. Complex. 3*, 406–428.

CYTRON, R., FERRANTE, J., ROSEN, B. K.,WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4, 451–490.

ERGIN, O., BALKAN, D., GHOSE, K., AND PONOMAREV, D. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, 304–315.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman.

GEORGE, L. AND APPEL, A. W. 1996. Iterated register coalescing. *ACM Trans. Program. Lang. Syst. 18,* 3, 300–324.

IRANI, S. AND LEUNG, V. 1996. Scheduling with conflicts, and applications to traffic signal control. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*. 85–94.

KNOBE, K. AND SARKAR, V. 1998. Array ssa form and its use in parallelization. In *Proceedings of the Symposium on Principles of Programming Languages*. 107–120.

KONDO, M. AND NAKAMURA, H. 2005. A small, fast and low-power register file by bit-partitioning. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Washington, 40–49.

LEE, J., MIDKIFF, S. P., AND PADUA, D. A. 1997. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing (LCPC'97).* 114–130.

LIPASTI, M. H., MESTAN, B. R., AND GUNADI, E. 2004. Physical register inlining. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, 325–335.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

PARK, J. AND MOON, S.-M. 2004. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst. 26,* 4, 735–765.

STEPHENSON, M., BABB, J., AND AMARASINGHE, S. 2000. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. 108–120.

TALLAM, S. AND GUPTA, R. 2003. Bitwidth aware global register allocation. In *Proceedings of the 30th Symposium on Principles of Programming Languages*. 85–96.

Received February 2013; accepted March 2013