Register Coalescing Techniques for Heterogeneous Register Architecture with Copy Sifting

MINWOOK AHN and YUNHEUNG PAEK
Seoul National University

Optimistic coalescing has been proven as an elegant and effective technique that provides better chances of safely coloring more registers in register allocation than other coalescing techniques. Its algorithm originally assumes homogeneous registers, which are all gathered in the same register file. Although this register architecture is still common in most general-purpose processors, embedded processors often contain heterogeneous registers, which are scattered in physically different register files dedicated for each dissimilar purpose and use. In this work, we show that optimistic coalescing is also useful for an embedded processor to better handle such heterogeneity of the register architecture, and developed a modified algorithm for optimal coalescing that helps a register allocator. In the experiment, an existing register allocator was able to achieve up to 13.0% reduction in code size through our coalescing, and avoid many spills that would have been generated without our scheme.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, compiler and optimization*

General Terms: Algorithms, Performance, Design, Experimentation

Additional Key Words and Phrases: Register allocation, register coalescing, compiler, embedded processors, heterogeneous register architecture

Part of this work was published in LCTES 2007. New contributions added to this article include Section 4.3, which describes the coloring heuristic which was applied to our modified optimistic coalescing; Sections 4.4 and 4.5, which propose two new techniques for further reducing spills in our modified optimistic coalescing; and Section 5, which extensively analyzes the impact of our coalescing technique with fuller benchmark codes.

This research is supported by the Korea Science and Engineering Foundation (KOSEF) NRL Program grant funded by the Korea government (MEST) (No. R0A-2008-000-20110-0) and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science, and Technology (MEST)/Korea Science and Engineering Foundation (KOSEF), grant number R11-2008-007-01001-0.

Authors' addresses: M. Ahn and Y. Paek, School of EECS/Center for SoC Design Tech, Seoul National Univ. #058, Kwanak, P.O. Box 34, Seoul 151-600, Korea; email: mwahn@optimizer.snu.ac.kr, ypaek@snu.ac.kr.

ACM Reference Format:
Ahn, M. and Paek, Y. 2009. Register coalescing techniques for heterogeneous register architecture with copy sifting. ACM Trans. Embedd. Comput. Syst. 8, 2, Article 16 (January 2009), 37 pages. DOI = 10.1145/ 1457255.1457263 http://doi.acm.org/10.1145/1457255.1457263

1. INTRODUCTION

The graph-coloring algorithm proposed by Chaitin [1982] is a popular heuristic for register allocation for traditional processors, adopted by almost all commercial compilers. It represents the interference relationship between the live ranges of symbolic variables in the application code as a graph called the interference graph (IG). In the IG, each node is checked if it can be trivially colored from the neighboring nodes.

A node is called significant-degree if the number of its neighbors is higher than that of the available colors, and otherwise called low-degree [Park et al. 2004]. If there is no available color, we "spill" a node, which means that it is removed from the IG in order to provide room for coloring the other nodes.

In the code generation of a compiler, many temporary variables and copy (or move) instructions are generated as the side effects of some analyses and optimizations. Many of these variables and copies are redundant, which only increase the overall code size as well as the number of spills during register allocation. As a result, register coalescing has been a key ingredient of register allocation as the technique that eliminate such copies. In the graph-coloring-based register allocation, register coalescing can eliminate copies by merging the copy related nodes in the IG. Two most well-known strategies for coalescing are optimistic coalescing [Park 2004] and iterated coalescing [George 1996]. According to an empirical study, optimistic coalescing generally shows better performance than the iterated counterpart; eliminating up to 20% more copy instructions [Park 2004]. This improvement is made possible because optimistic coalescing, in its first phase, can eliminate all coalescible copies by employing an aggressive coalescing scheme [Chaitin 1982], which aggressively merges every copy related node in the IG. But, the aggressive scheme alone sometimes adversely affects the coloring of the IG. That is, when we merge the copy-related nodes in the IG, the degree of the merged node may be increased, which would make it impossible to color the merged node, and might produce a large number of spills in the end. To minimize this malicious effect of aggressive coalescing on graph coloring, optimistic coalescing subsequently applies an additional scheme, called live range splitting, which splits the merged nodes in the IG and, checks the possibility of separately coloring the split nodes. This added scheme enables optimistic coalescing to produce a less number of spills than mere aggressive coalescing.

Embedded processors often have complex, irregular architectures that are customized to optimize the performance and the energy efficiency of certain applications. This irregularity usually makes the coalescing for these processors more challenging than for conventional general-purpose processors (GPPs) with relatively regular architectures. One such example is an architecture with multiple register files that are physically distributed to different functional units in

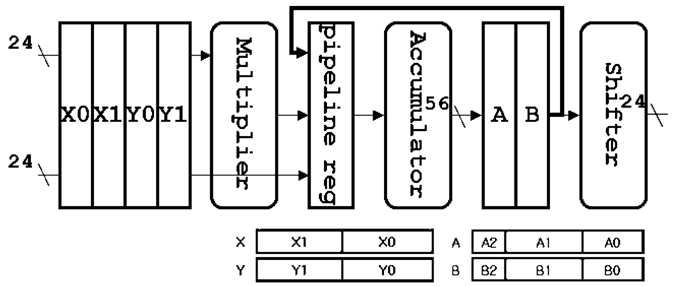


Fig. 1. Partial diagram of DSP563xx

the processor. We name this, a heterogeneous register architecture, in the point of view that the accessibility of its registers to each functional unit is limited by the topology of the data path inside the processor, and hence the use of each register file is differentiated in the code. DSP56k, ADSP21xx, TMS320C2xxx, and DSP563xx are the examples of the heterogeneous register architectures. Figure 1 shows the data-path of Freescale DSP563xx, which has two accumulator registers (A, B) connected to the output of the ALU. Four input registers (X0, X1, Y0, Y1) are used as input operands of the multiplier or ALU.

Heterogeneous register architectures are not common in GPPs, which usually have registers gathered in a single common register file. We view these GPP registers as homogeneous registers, since they are all accessible to the same functional unit in the processor, thus being used interchangeably for the same purpose. In general, register coalescing has greater significance for heterogeneous register architectures than for homogeneous ones because the compiled code for processors with heterogeneous registers has more copy instructions than that for those with homogeneous registers since extra copy operations are needed to move data between different register files to execute ALU instructions that demand registers in specific files as their operands. For instance, according to [Zivojnovic 1994], about 55% of instructions in the code for the Motorola DSP56k processor are copies, which are unusually high, as compared to the case of GPPs. This fact indicates that the code quality in a heterogeneous register architecture would rely on how efficiently to minimize such copy instructions [Araujo 1998]. Unfortunately, the register coalescing problem is more complex for the heterogeneous register architecture than the homogeneous one. In the homogenous architecture, it is relatively trivial to merge copy related nodes in the IG because all the registers that can be assigned to those nodes belong to a single file, and any register in the file can be allocated to the merged node. In contrast, in the heterogeneous architecture, each node of the IG may belong to different register files. Thus, we should decide a register file in which a register is allocated to that merged node. Likewise, we should take care of the colorability of each split node after live range splitting. Also in the homogeneous architecture, all registers are in the same file, so we only check the degree of each node for the colorability of that node. But, in the heterogeneous one, the colorability is also influenced by the decision of the register files where the registers allocated to each node belong.

In this article, we propose an extended version of existing optimistic coalescing algorithm to cope with the heterogeneous register architectures. Section 2 explains a generalized graph-coloring algorithm for register allocation targeting heterogeneous register architectures, which we use as the basis of our register allocation framework in this work. Section 3 compares iterated and optimistic coalescing schemes in principle. Then, Section 4 describes how we extend the existing optimistic coalescing scheme to handle heterogeneous register architecture and proposes two techniques for reducing more spills in the extended optimistic coalescing scheme. Section 5 presents our experimental results that demonstrate where our register allocator implemented with the extended optimistic coalescing scheme outperforms an existing register allocator by Smith et al. [2004] for heterogeneous register architectures.

2. REGISTER ALLOCATION FRAMEWORK

Several studies (Kong et al. [1998], Scholz et al. [2002], Koes et al. [2005], Lee et al. [2006], Daveau et al. [2004]) on the exploitation of the register resources have been conducted to handle the irregularity of heterogeneous register architectures. Due to the extreme complexity of the register allocation for such architectures, Kong et al. [1998], Scholz et al. [2002], and Koes et al. [2005] resort to expensive algorithms based on the integer linear programming or others with exponential-time complexities. To solve this problem practically in polynomial time, Smith et al. [2004], Lee et al. [2006], and Daveau et al. [2004] do register allocation based on the traditional graph coloring scheme. LEE et al. [2006] use some cost model for improving copy propagation. Daveau et al. [2004] deal with the heterogeneity of the registers based on the Briggs' style register allocation [1992]. Smith et al. [2004] suggest a generalized graph-coloring algorithm that handles both homogeneous and heterogeneous architectures. In our work, we use basically the same algorithm as Smith's with additional modifications to fit into our compiler [Ahn et al. 2007], where we also add our optimistic coalescing technique proposed in this paper. In this section, we describe the register allocation algorithm framework.

2.1 Register Class

In our register allocation framework, heterogeneous registers are grouped by logical units called *register classes*. The term "register class" has been frequently used by others [Smith et al. 2004; Liem et al. 1994; Paulin et al. 1994; Feuerhan 1988; Stallman 1994]. Although the name is the same, its definition differs depending on its usage in each work. To avoid confusion with others, we formally define our notion of register classes in this section.

Definition 1. Given a processor P, let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of all instructions defined on P, and $R = \{r_1, r_2, \ldots, r_m\}$ be a set of all its registers. For instruction $i_j \in I$, we define a set of all its operands, $op(i_j) = \{O_{j1}, O_{j2}, \ldots, O_{jk}\}$. Assume $\phi_l(i_j)$ is a set of all the registers that can appear at the position of some operand O_{jl}, $1 \leq l \leq k$. Then, we say that $\phi_l(i_j)$ forms a **register class** for i_j.

Our definition of register class is rather logical, as being classified according to their usages in the instruction set rather than physical layouts in the hardware. By Definition 1, any member in the register class for a given instruction can be interchangeably referenced as an operand of the instruction. In this sense, we view all registers in the same class as being homogeneous in terms of their roles in the code generation. For instance, the ARM machine has an instruction, **add *reg₁,reg₂,reg₃*,** where any of its sixteen registers (r0,r1,. . . ,r15) can appear as any operand in the instruction; that is,

$$\phi_1(\mathbf{add}) = \phi_2(\mathbf{add}) = \phi_3(\mathbf{add}) = \{\mathrm{r0}, \mathrm{r1}, \dots, \mathrm{r15}\}$$

So, this set forms a single register class for **add** according to Definition 1. This means that all the 16 registers in ARM are homogeneous for **add**. As another example in Figure 1, consider the instruction **mpy s_1,s_2,d** of DSP563xx, which multiplies the first two sources and places the product in the destination. DSP563xx restricts s_1 and s_2 to be four registers **X0**, **X1**, **Y0** or **Y1**, and d to be the upper half of **A** or **B** (i.e., **A1** or **B1**). Consequently two register classes {X0,X1,Y0,Y1} and {A1,B1} are dedicated to **mpy**, respectively at s_1/s_2 and at d. From this, we see that unlike in the case of **add**, not all registers in DSP563xx are equally usable by **mpy**. So, we here say that the DSP563xx registers are nonhomogeneous (or in other words, heterogeneous) for **mpy**.

Definition 2. Using Definition 1, we define $\Phi(i)$ a collection of distinct register classes for instruction i as follows:

$$\Phi(i) = \bigcup\nolimits_{l=1}^{k} \{\phi_l(i)\}.$$

We say that two instructions, i and j, have **disjoint** register classes if $\Phi(i) \cap \Phi(j) = \emptyset$.

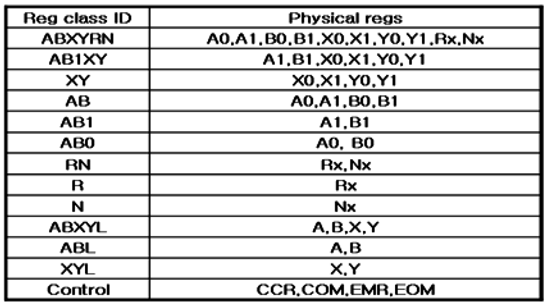
Definition 3. For processor P with instruction set $I = \{i_1, i_2, .., i_n\}$, the whole collection of register classes, denoted by Φ^P, is defined:

$$\Phi^P = \bigcup\nolimits_{j=1}^{n} \Phi(i_j)$$

By Definition 2, we have $\Phi(\mathbf{add}) = \{\{\mathrm{r0}, \dots, \mathrm{r15}\}\}$ for instruction **add** in the earlier example. In ARM, for virtually all other ALU instructions i, we have $\Phi(i) = \Phi(\mathbf{add})$. This equivalently means that Φ^{ARM} contains only one register class consisting of the whole sixteen registers. In this sense, we regard the register architecture of ARM as being homogeneous. In case of DSP563xx, however, different registers are usually dedicated to the machine instructions, which make the processor partially homogeneous only in the subsets of machine instructions. For example, recall that even a single instruction **mpy** in DSP563xx, has two different sets of homogeneous registers. In fact, Φ^{DSP563xx} is listed in Figure 2, from which we can recognize that the register architecture of DSP563xx is heterogeneous.

2.2 Register Aliasing

Multiple register names may be aliased for a single hardware register in various modern architectures [Smith et al. 2004]. For instance, Figure 1 shows that double-word registers **X** and **Y** of DSP563xx are divided into two subregisters, each of which can be also referenced as an individual operand. So, the same



Reg class ID	Physical regs
ABXYRN	A0,A1,B0,B1,X0,X1,Y0,Y1,Rx,Nx
AB1XY	A1,B1,X0,X1,Y0,Y1
XY	X0,X1,Y0,Y1
AB	A0,A1,B0,B1
AB1	A1,B1
AB0	A0, B0
RN	Rx,Nx
R	Rx
N	Nx
ABXYL	A,B,X,Y
ABL	A,B
XYL	X,Y
Control	CCR,COM,EMR,EOM

Fig. 2. Register classes of DSP563xx.

physical register is accessible via three different names **X0**, **X1**, and **X**. This relationship between registers is called *register aliasing*. The register aliasing makes the register allocation a little bit complicated because if we assign one register to a node in the IG, we cannot assign the aliased registers to its adjacent nodes. As the register aliasing is also often found in the heterogeneous register architecture, it is crucial to consider it in the register allocation.

2.3 Generalized Graph Coloring Algorithm

Chaitin [1982] assumes a single register class architecture where all registers are homogeneous. Under this assumption, it reduces register allocation to a k-coloring problem where k homogeneous registers for m variables are represented as k distinct colors on m nodes in the IG. However, this k-coloring problem is no longer directly applicable to the heterogeneous register architecture such as DSP563xx. To explain this, consider the IG for four variables in Figure 4. First of all, in conventional k-coloring, every node in the graph is assumed to be the same k-colorable. However, to handle heterogeneous registers, we must allow a different colorability for each node. For instance, if ϕ_x denote a register class bound to variable x, we see from Figure 2 that $\phi_{\mathbf{t3}}$ contains two registers. This implies that node **t3** in Figure 4 is 2-colorable. On the other hand, node **t1** is 4-colorable since $\phi_{\mathbf{t1}}$ has four registers. In ordinary k-coloring, every node must share the k resources of the same types, which is not always true for the heterogeneous register architecture. As an example, notice that the colors of **t1** are {**A,B,X,Y**} while the colors of **t3** are {**A1,B1**}. Another complexity imposed by the heterogeneous register architecture on register allocation is that each node has different impacts on the colorability of its neighbors in the IG due to the register aliasing. For instance, suppose register **A** is assigned to node **t1**. Then, this coloring decision will strip two colors off node **t4** since **t4** can be no longer assigned register **A0** or **A1**. However, at the same time, it strips only one color (i.e., **A1**) off node **t3**.

Based on all these observations on the complexities of register allocation for the heterogeneous register architectures, Smith et al. [2004] introduce a new measure, called the squeeze, which supplements the notion of k-colorability in graph coloring. Since register allocation is an intrinsically intractable problem, conventional k-coloring algorithms have used a heuristic, called the degree-less-than-k, based on the colorability information of each node, and reduce the

register	set of alias registers
X	X, X0, X1
X0	X, X0
X1	X, X1
Y	Y, Y0, Y1
Y0	Y, Y0
Y1	Y, Y1
A	A, A1, A0
A1	A, A1
A0	A, A0
B	B, B1, B0
B1	B, B1
B0	B, B0

Fig. 3. Alias relation of DSP563xx.

problem size substantially, thereby achieving a practically tractable computation time. Smith et al. [2004] also devise a similar heuristic technique based on the squeeze in their algorithm.

Suppose there is a variable node x in an interference graph G and a register class ϕ_x is bound to x. Then, the squeeze for x, denoted by σ^*_x, is defined as the maximum number of registers from ϕ_x that cannot be allocated to x due to an assignment of registers to all its neighbors. To formally define σ^*_x, assume that ϕ_x has k registers, implying that x is initially k-colorable. Now, suppose a neighbor y of x is assigned a register $\mathbf{r}$ from its class ϕ_y. Then, if $\mathbf{r}$ aliases with k' registers in ϕ_x, the initial colorability of x will reduce by k' due to its neighbor y because by the definition of alias, the k' registers could not be allocated to x anymore. As a result, x would become at most (k-k')-colorable after $\mathbf{r}$ is allocated to y. This can be summarized as follows:

$$\text{reduction of colorability of } x \text{ due to } y = k' = |A(\mathbf{r}) \cap \phi_x|.$$

where $A(\mathbf{r})$ denotes the set of all aliases of $\mathbf{r}$. Likewise, the remaining neighbors are all assigned registers from their classes, forming a set of registers assigned to x's neighbors, say R, as a result. Now, let k'' be the total amount of reduction of x's colorability due to all its neighbors. Then, obviously, we have:

$$k'' = \left|A(R) \cap \phi_x\right| \;\text{ for } A(R) = \bigcup_{\forall r \in R} A(r).$$

For every coloring of x's neighbors, we would have many different configurations for R. The squeeze σ^*_x is defined for the set $\Re$ of all colorings of x's neighbors as follows:

$$\sigma^*_x = \max_{\forall R \in \Re} \left(\left|A(R) \cap \phi_x\right|\right)$$

From the result, we conclude that node x would always be at least (k-σ^*_x)-colorable with any coloring of its neighbors. This finding induces a heuristic that x can be immediately removed from G as long as $k > \sigma^*_x$ since it is then trivially colorable under any circumstance. For instance, in Figure 4, to compute the squeeze $\sigma^*_{\mathbf{t4}}$ for node **t4**, we identify all possible colorings R^* of its neighbor **t1**. From Figure 2, we find four possible colorings: that is, $\Re = \{\{\mathbf{A}\},\{\mathbf{B}\},\{\mathbf{X}\},\{\mathbf{Y}\}\}$. Then, from Figure 3, we identify the aliases of each coloring as follows:

$$\begin{aligned} A(\mathbf{A}) &= \{\mathbf{A}, \mathbf{A0}, \mathbf{A1}\}, A(\mathbf{B}) = \{\mathbf{B}, \mathbf{B0}, \mathbf{B1}\}, \\ A(\mathbf{X}) &= \{\mathbf{X}, \mathbf{X0}, \mathbf{X1}\}, A(\mathbf{Y}) = \{\mathbf{Y}, \mathbf{Y0}, \mathbf{Y1}\}. \end{aligned}$$

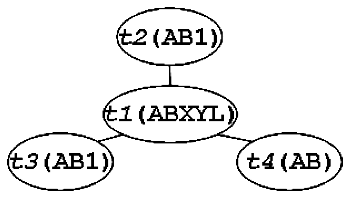


Fig. 4. Interference graph and the calculation of squeeze.

Using this information, we compute for $\phi_{\mathbf{t4}} = \{\mathbf{A0},\mathbf{A1},\mathbf{B0},\mathbf{B1}\}$:

$$\begin{aligned}\sigma^*_{\mathbf{t4}} &= max(|A(\mathbf{A}) \cap \phi_{\mathbf{t4}}|, |A(\mathbf{B}) \cap \phi_{\mathbf{t4}}|, |A(\mathbf{X}) \cap \phi_{\mathbf{t4}}|, |A(\mathbf{Y}) \cap \phi_{\mathbf{t4}}|)\\ &= max(2, 2, 0, 0) = 2.\end{aligned}$$

Node **t4** is initially 4-colorable since $|\phi_{\mathbf{t4}}| = k = 4$. As a result, we have $k - \sigma^*_{\mathbf{t4}} = 2$. This means that even after **t1** is assigned any register from its class $\phi_{\mathbf{t1}}$, **t4** would remain at least 2-colorable. Here, we therefore safely remove node **t4** from the interference graph of Figure 4. In the same manner, we identify the remaining nodes are trivially colorable, hence removing all from the graph. Our experiments show that, similar to the degree-less-than-k heuristic, this new squeeze-based heuristic is also effective to simplify the register allocation for the heterogeneous register architecture by precluding many trivially-colorable nodes.

Unfortunately, finding an ideal squeeze σ^* is virtually impossible for a large interference graph because the set $\Re$ has exponential size complexity in the number of nodes. For instance, in Figure 4 to compute $\sigma^*_{\mathbf{t1}}$ for node **t1**, we should consider $16(= 2 \times 2 \times 4)$ cases of colorings for its three neighbors even for such a small graph. So, in practice, an approximated squeeze σ [Smith et al. 2004] is sought to prevent the compilation time from increasing dramatically. To compute σ_x for node x in a graph G, assume the followings:

1. All register classes bound to x's neighbors are classified into m distinct register classes $\pi_i, 1 \leq i \leq m$.
2. $n_i = |\{y \,|\, y \in N \wedge \phi_y = \pi_i\}|$ where N is the set of x's neighbors.

Then, we define σ_x as follows:

$$\sigma_x = \sum_{\forall \pi_i, 1 \leq i \leq m} n_i \cdot \max_{\forall r \in \pi_i}(|\phi_x \cap A(r)|)$$

This formula guarantees that σ is a safe approximation of σ^*. However, it is sometimes too conservative, thus hindering some trivially colorable nodes from being removed from G. In Figure 4 for example, $\sigma^*_{\mathbf{t1}}$ is 2, implying that **t1** is always at least 2-colorable since it is initially 4-colorable. But, $\sigma_{\mathbf{t1}}$ evaluates to:

$$\sigma_{\mathbf{t1}} = \max_{\forall r \in \mathbf{AB}}(|\phi_{\mathbf{t1}} \cap A(r)|) + 2 \cdot \max_{\forall r \in \mathbf{AB0}}(|\phi_{\mathbf{t1}} \cap A(r)|) = 1 + 2 = 3.$$

This estimated value $\sigma_{\mathbf{t1}}$ falsely informs us that **t1** is only 1-colorable. To prevent σ from being too overly estimated, as in this case, other safeguards such as the upper bound are applied [Smith et al. 2004].

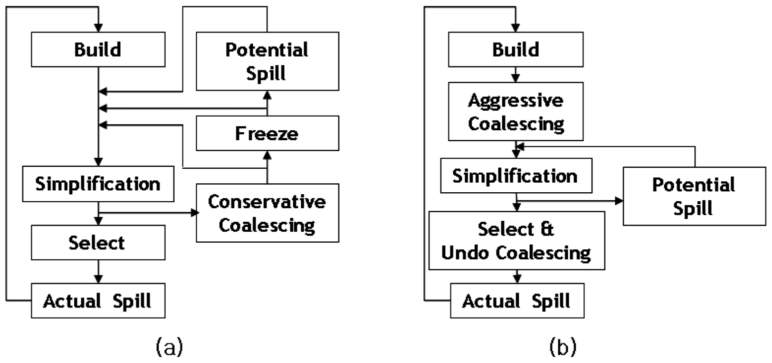


Fig. 5. (a) Iterated coalescing and (b) optimistic coalescing

3. ITERATED VS. OPTIMISTIC COALSECING

Iterated coalescing uses a conservative coalescing scheme. In the conservative scheme, two copy-related nodes **x** and **y** in the IG are coalesced only when the original colorability of the IG is not affected by the coalescing. This can be ensured by testing if their coalesced node **xy** has less than k significant-degree neighbors, which equivalently means that **xy** is low-degree. Conservative coalescing guarantees that it does not produce any spills from coalescing. But it sometimes misses the further chance of coalescing [George et al. 1996]. As illustrated in Figure 5(a), iterated coalescing attempts to overcome this drawback by iteratively applying both conservative coalescing and simplification phases. The simplification phase prunes the low-degree, non-copy-related nodes from the IG, and reduces the degree of copy-related nodes, yielding more opportunities for coalescing in the conservative coalescing phase. These two phases are repeated until there are left only significant or copy-related nodes in the IG. If neither phase can be applicable, low-degree copy-related nodes are frozen, which means giving up a further chance of its coalescing by removing its copy-related edges and marking it as a non-copy-related node. As frozen nodes are no more a copy-related nodes, they may be pruned in the simplification phase (see Figure 5(a)) [George et al. 1996].

Although iterated coalescing usually eliminates many more copy instructions than conservative coalescing without introducing spills, it still has some weaknesses. One is that it gives up the chances of coalescing too early even if a coalescible node violating the colorability criterion is not necessarily spilled. If a coalesced node violates the criterion without being actually spilled, the decision for spill could be delayed, which provides room for further coalescing [Park et al. 2004]. Moreover, like conservative coalescing, iterated coalescing is too conservative to enjoy the positive impact of coalescing. To explain this, consider the IGs in Figure 6, where nodes **a** and **c** are copy-related. Notice that in Figure 6(a), these nodes have a common neighbor **b**, while in Figure 6(b), each of them respectively has different neighbors **b** and **d**. As shown here, if copy-related nodes have a common neighbor, coalescing brings positive impact on coloring, but otherwise, it brings negative impact. In Figure 6(a), see that the IG remains 2-colorable with fewer nodes after coalescing. In contrast, in

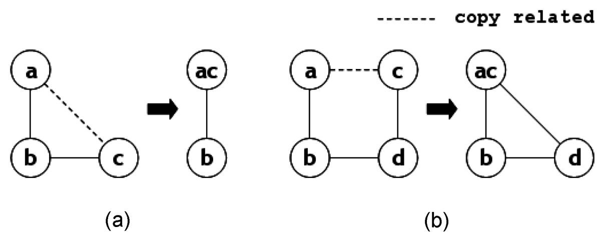


Fig. 6. Examples of (a) positive and (b) negative impact of coalescing.

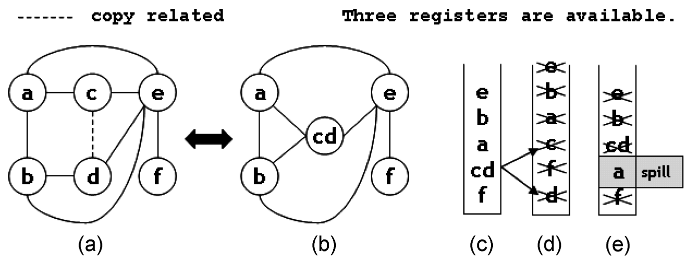


Fig. 7. An example of the live range splitting.

Figure 6(b), the IG turns from 2-colorable to 3-colorable. The aggressive coalescing scheme attempts to overcome this weakness of conservative coalescing by fully exploiting this positive impact of coalescing. Park et al. [2004] have shown in their experiments that the aggressive scheme usually better colors the IG than a conservative scheme. As shown in Figure 5(b), optimistic coalescing employs the aggressive scheme. We have observed that taking such advantage of aggressive coalescing, it outperforms iterated coalescing in most cases. In optimistic coalescing, the decision of spills for coalesced nodes is delayed after coalescing.

Although the negative impact of coalescing is negligible, the aggressive scheme should be applied with caution because once the coalesced node is actually spilled, it may trigger producing many other spills. This is due mainly to the fact that the coalesced nodes in the IG create variables with long live ranges in the code. This detrimental effect can be reduced by live range splitting [Park et al. 2004], which splits a long live range into short ones by inserting copy/move or load/store instructions. If the live range of a variable becomes shorter by live range splitting, it will surely increase the chance of avoiding spills since a shorter live range would less likely conflict with other ranges. Registers are allocated for the split live ranges of the variable. To minimize spills, we must carefully decide where the live range of the coalesced node is split. According to Park et al. [2004], the original location of a copy instruction before coalescing its copy-related nodes is the good place for splitting. This in fact corresponds to the undo of coalescing. This undo splits the coalesced node back into its original copy-related nodes, recovering the original IG. See that the IG in Figure 7(b) is restored after splitting the coalesced node **cd.** In Figure 7(a), the degree of split nodes **c** and **d** is lower than that of **cd**, which makes it easy to color the split nodes.

As shown in Figure 5(b), optimistic coalescing merges all the copy-related nodes (**c**,**d**) before simplification, resulting in the IG of Figure 7(b). Suppose there are three available registers. Then, following the conventional register allocation algorithm, node **f** can be pruned and pushed into the bottom of a stack in Figure 7(c), where all nodes except **f** are significant-degree nodes. In this case, optimistic coalescing chooses the coalesced node **cd** as a candidate of spill (*potential spill*) due to live range splitting. This choice enables us to prune three nodes (**a, b, e**) from the IG and push them into the stack of Figure 7(c). Now that there is no node left in the IG, we pop nodes from the stack and color each of them. Nodes **e, b** and **a** can be trivially colored. The coalesced node **cd** has no available color since the interfering nodes **e**,**b**, and **a** preoccupied all available colors. In this case, optimistic coalescing applies live range splitting to restore the IG into the original one in Figure 7(a). Then, all nodes can be colored as shown in Figure 7(d). However, if we do not apply live range splitting to the IG in Figure 7(b), we would arbitrary choose for the potential spill a different node instead of **cd**. Suppose we choose node **a**. Then, the resulting stack should be the one in Figure 7(e), and node **a** becomes a real spill (*actual spill*).

If all of the split nodes from live range splitting are significant-degree nodes, they are marked as actual spills. If there are any colorable nodes among them, optimistic coalescing decides the coloring of the nodes in a way of minimizing spills. This decision should be made carefully to prevent the integrity of coloring the other nodes after splitting. To explain this, notice in Figure 5(b) that the undo coalescing phase that performs live range splitting follows the simplification phase because, right after simplification, we can identify what nodes in the IG are potential spills. In this phase, all nodes in the IG are colored in the reverse order that they were pruned from the IG and pushed into a stack, as described above. The nodes under a coalesced node in the stack might be pruned from the IG as their degrees were decreased after coalescing. So, if the coalesced node is split, the degrees of some nodes under the node may be altered, which makes the coloring of all nodes ruined. Optimistic coalescing prevents this by coloring only one of the split nodes.[1] That is, when a coalesced node is split into multiple nodes, only one split node is left in the original position in the stack, and the others are all placed at the bottom of the stack to delay their coloring decision till the end. In this way, we do not change the coloring of all the nodes below the coalesced node. Thus, it is now safe to color the other split nodes later if there are available colors at the time. In Figure 7(c), the coalesced node **cd** is split. One of the split node, **c** is colored first, but the other node **d** is colored later than node **f** in the stack, as shown in Figure 7(d).

4. OPTIMISTIC COALESCING FOR HETEROGENEOUS REGISTERS

The coalescing techniques described in Section 3 are originally designed to work with the traditional graph-coloring algorithm for an architecture with homogeneous registers. Therefore, despite the advantages of optimistic coalescing,

[1]If there are multiple colorable nodes among the split nodes, optimistic coalescing chooses a node with the highest spill cost to minimize spills.

its existing technique cannot be directly used in our register allocation framework because, as explained in Section 2, the graph-coloring algorithm in our framework has been extended to support heterogeneous registers, so the existing technique cannot deal properly with unique constraints enforced by heterogeneous register architectures incorporated in the extended graph coloring algorithm. In this section, we discuss how we have modified the optimistic coalescing technique to cope with heterogeneous registers. This requires two additional considerations related to coalescing copy-related nodes and splitting the live ranges of coalesced nodes in the IG. Furthermore, as we will show you in our experiments, our extended optimistic coalescing scheme produces more spills in some benchmark codes. We have discovered that these are the cases where the negative impact of the aggressive coalescing strategy in optimistic coalescing overwhelms the merit of optimistic coalescing despite of those considerations; therefore, our modified optimistic coalescing inserts many spills. We propose two additional techniques for decreasing the spills. In the following subsections, we discuss each of them.

4.1 Coalescing Copy-Related Nodes with RCRT

In our register allocation framework, a register name can be bound to multiple register classes. If copy-related nodes are bound to different register classes, the register class of their coalesced node should be the intersection of their register classes. To compute the intersection of register classes, we should know the relationship between the register classes defined in the target architecture. To summarize this relationship, we propose a table called, the register class relation table (RCRT) in this work. Figure 8 shows the RCRT of the DSP563xx constructed from the register class information in Figure 2 and Figure 3. This table represents the inclusion relationship between the register classes of DSP563xx. For instance, the intersection of the register class AB1XY and AB1 is the register class AB1. Besides the intersection of two register classes, this table shows two special attributes **a** and **d**. The attribute **a** denotes that each of the two register classes contains the registers whose aliased registers belong to the opposite register class. We use **d** to denote the two disjoint register classes explained in Section 2.

In optimistic coalescing, all copy-related nodes are aggressively coalesced before simplification. As we noted above, if the register classes of the copy-related nodes in the IG are different, we should determine whether coalescing is possible or not. Even if it is possible, we also need to determine the register class of the coalesced node. There are three cases in this decision process. First, if the two register classes are disjoint, we cannot coalesce them because there is no common registers to allocate in the source operand and the destination operand of the copy instruction. We can check this by using the RCRT. Second, we do not coalesce the copy-related nodes whose register classes are aliased. In the aliased register classes, there is a common physical register to be allocated. but this register may not be explicitly shown in both register classes. For example, Figure 8 shows that two classes AB and ABL are aliased. Notice that the registers in AB is {A0, A1, B0, B1} and those in ABL are {A, B}.

a : Aliased d : Disjoint

	ABXYRN	ABXYL	NR	AB	AB1XY	XYL	N	R	AB0	AB1	ABL	XY
ABXYRN	ABXYRN	a	NR	AB	AB1XY	a	N	R	AB0	AB1	a	XY
ABXYL	▨	ABXYL	d	a	a	XYL	d	d	a	a	ABL	a
NR	▨	▨	NR	d	d	d	N	R	d	d	d	d
AB	▨	▨	▨	AB	AB1	d	d	d	AB0	AB1	a	d
AB1XY	▨	▨	▨	▨	AB1XY	a	d	d	d	AB1	a	XY
XYL	▨	▨	▨	▨	▨	XYL	d	d	d	d	d	a
N	▨	▨	▨	▨	▨	▨	N	d	d	d	d	d
R	▨	▨	▨	▨	▨	▨	▨	R	d	d	d	d
AB0	▨	▨	▨	▨	▨	▨	▨	▨	AB0	d	a	d
AB1	▨	▨	▨	▨	▨	▨	▨	▨	▨	AB1	a	d
ABL	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	ABL	d
XY	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	XY

Fig. 8. Register class relation table (RCRT) of DSP563xx.

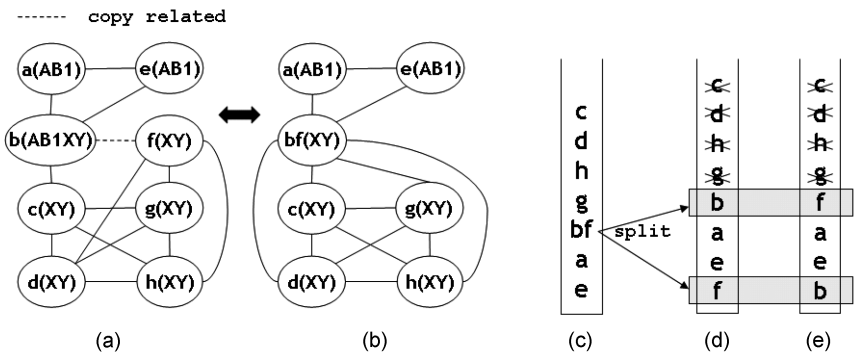


Fig. 9. An example of coalescing and live range splitting.

Although A0, A1, B0, and B1 are the upper and lower half subregisters of A and B, respectively, there is no common register name to allocate. Third, if in RCRT we find an intersection of two register classes of the copy-related nodes, we can coalesce them. In this case, the register class of the resulting node is the intersection of the register classes. The copy-related nodes b and f in Figure 9(a) have different register classes. The register class of b is AB1XY and that of f is XY. Figure 8 gives that the intersection of the register classes AB1XY and XY is XY. Figure 9(b) shows the resulting IG after coalescing including their coalesced node bf with the intersected class XY. In the example of Figure 8, one register class always subsumes the other when two register classes are intersected. If not, a new register class is temporarily made during the compilation, and the RCRT is updated, which is different from the previous work of Smith [2004]. In their work, they should manually modify the register allocator in order to deal with this case.

4.2 Splitting Live Range with New Spill Metric

After merging all copy-related nodes, optimistic coalescing prunes nodes from the IG if the nodes are trivially colored. If nodes remaining in the IG are not trivially colorable any more, the register allocator selects a node for a potential spill. This decision for the potential spill is based on the spill metric of every node in the IG. The spill metric, denoted by $M(v)$, of node v[2] in the IG is calculated from the following equation:

$$M(v) = C(v)/D(v) \tag{1}$$

where $C(v)$ is the spill cost of v, and $D(v)$ is the degree of v in the IG. The spill cost of v is computed as:

$$C(v) = w_d \times \sum_{def \in v}^{1} 0^{d(def)} + w_u \times \sum_{use \in v}^{10^{d(use)}}$$

where w_d is the relative weight of the definition in v, w_u is the relative weight of the use in v, and $d(x)$ is the depth of the loop nest at the location of x in the application code [Chaitin 1982]. $C(v)$ indicates how much the performance is degraded by the spill of v. So does $M(v)$, since it is proportional to $C(v)$, but notice that $M(v)$ is inversely proportional to $D(v)$. This is due to the simple fact that the more interfering nodes a node has in the IG, the greater chance of spilling the node opens up for coloring other nodes. Consequently, a node with a minimum spill metric is selected for a potential spill since it helps the coloring of as many neighboring nodes as possible with the minimum performance degradation due to the insertion of load/store instructions from spills.

Recall in Section 2 that our register allocator for heterogeneous register architectures determines the colorability of node v by the value of its squeeze σ_v. As $\sigma_v$is the maximum number of registers from the register class ϕ_v of node v that could be denied to v due to an assignment of registers to the current neighbors of v, v is trivially colorable if $\sigma_v < |\phi_v|$. Likewise, the decision for the potential spill is influenced by not only the degree of a node in the IG but also the register class bound to the node. For instance, in Figure 9(b), after two nodes a and e were pruned from the IG, all remaining nodes (bf, c, d, g, h) cannot be pruned because there is no node whose squeeze is less than the size of its register class. Recall that optimistic coalescing prefers a coalesced node than a noncoalesced one for a potential spill. Thus, in Figure 9(b), node bf is selected for a potential spill and pruned from the IG. Then, the four nodes (c, d, g, h) could be pruned. The pruned nodes are piled inside a stack (see Figure 9(c)).

As being popped from the stack later for coloring, each node is assigned a register among the registers in the register class bound to the node. Assume that as the four nodes (c, d, h, g) at the top of the stack are popped, they are respectively assigned the registers {X0, X1, Y0, Y1}. Then, we can clearly see that the coalesced node bf has no register available for allocation since its squeeze $\sigma_{\mathrm{bf}}(= 4)$ is identical to the size of the register class $|\phi_{XY}|(= 4)$ bound to bf. To escape from this dead end, we follow the original scheme of optimistic coalescing in Section 3; that is, we try a second chance for the coloring of bf by restoring the original IG of Figure 9(a) via live range splitting. Now, nodes b and

[2]A node in the IG represents a live range of a symbolic variable in the code. So we use the terms node in the IG, symbolic variable, and live range interchangeably without clearly differentiating their meanings.

f in the IG are colorable since both σ_b and σ_f are less than the size of their register classes AB1XY and XY. Therefore, we color one of the split nodes immediately, and delay the coloring of the other node by placing it at the bottom of the stack, as in the case of either Figure 9(d) or (e). The question here is which node should be selected for immediate coloring. Recall in Section 3 that among all split nodes, the original scheme of optimistic coalescing would select the node v with the maximum value of the spill metric $M(v)$, above shown in Equation (1). For homogeneous register architectures, this simple metric should be workable since any register can be interchangeably assigned to any one among the split nodes. But, for heterogeneous architectures, a more elaborate spill metric that reflects the heterogeneity of registers allocatable to the split nodes is required to avoid creating unnecessary spills. To illustrate this, assume $C(\mathrm{b}) = C(\mathrm{f})$ in the example of Figure 9. Then, in the original optimistic coalescing scheme, the spill metrics of nodes b and f is obviously the same. So the scheme would like to break the tie by coloring arbitrarily first either b or f. Suppose b is first colored with A1 or B1 as shown in Figure 9(d), then there might be an actual spill when a and e are being assigned registers. On the other hand, if f is first colored with X0 (see Figure 9(e)), there would be no spill. Any registers in AB1XY except those assigned to c, a, and e can be allocated to b.

The different outcome of register allocation between these two cases is caused by the fact that both split nodes are bound to different register classes with different sizes, which does not likely occur in a homogeneous register architecture. To manage this characteristic of heterogeneous register architectures, we suggest a heuristic that gives the first priority for coloring to a node bound to a register class with the smallest size. This is of course based on our observation that the other split nodes with larger-sized register classes usually have more chance to be colored later. In the above example, the register class XY bound to node f is a proper subset of the class AB1XY bound to node b, so we have

$$|\phi_{XY}| = 4 < |\phi_{AB1XY}| = 6,$$

implying that two more registers are allocatable to b than to f. Thus, coloring f first would likely increase the chance to avoid extra spills, which actually turns out to be true, as discussed above.

This example has motivated us to modify the existing spill metric so as to reflect such effect of the register class on optimistic coalescing for heterogeneous register architectures. For this purpose, we introduce a value, called the *register class influence* $\varepsilon(\phi_v)$, which measures the amount of influence a node v bound to a register class ϕ_v has on the coloring of its neighboring nodes. From the fact that σ_x is proportional to $|\phi_y|$ (y is a neighboring node of node x in an IG), we can derive a relation of the register class influence between two nodes v and u respectively bound to register classes ϕ_v and ϕ_u:

$$\varepsilon(\phi_v) \leq \varepsilon(\phi_u) \quad \text{for} \quad \phi_v \subseteq \phi_u.$$

For instance, in Figure 9(a), since the register class XY of node f is a subset of the class AB1XY of node b, the neighbors of b would more likely have larger squeezes than those of f. This implies that the colorability of b's neighbors should be more limited than that of f's neighbors, thus resulting in $\varepsilon(\phi_f) \leq \varepsilon(\phi_b)$. This

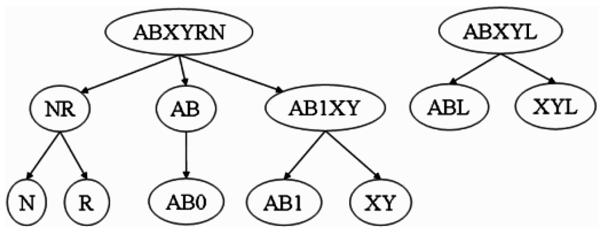


Fig. 10. Register class tree (RCT) of DSP563xx.

relation can be visualized by the *Register Class Tree* (RCT) where the nodes are all register classes of the target architecture, and the edges represents the inclusion relation between the classes. In the tree, a register class ϕ is a descent of another class ϕ' if $\phi \subset \phi'$. If there is no other class ϕ'' such that $\phi \subset \phi'' \subset \phi'$, then there will be an edge between ϕ and ϕ' in the RCT. Figure 10 shows an example of the RCT from the RCRT in Figure 8. Notice that two register classes ABL and XYL are the children of the class ABXYL since they are subsets of ABXYL. We can see from the figure that the RCT built for the target processor can be actually a forest (i.e., a collection of multiple trees).

The depth $\delta(\phi)$ of a register class ϕ in an RCT is the distance from the root node of a tree to ϕ, assuming the depth of the root node is one. For instance in Figure 10, the depth of ABXYRN is 1 and that of XY is 3. Based on the argument above, we conclude that $\delta(\phi)$ is inversely proportion to $\varepsilon(\phi)$, resulting in the equality:

$$\varepsilon(\phi_v) = \frac{1}{\delta(\phi_v)}$$

where ϕ_v is the register class of a node v in an IG. In the example of Figure 10, we have

$$\varepsilon_{ABXYRN} = 1 \quad \text{and} \quad \varepsilon_{XY} = 1/3.$$

Now, we modify the original spill metric in Equation (1) by taking into account the influence ε of the register class ϕ_v of each node v on the coloring. The degree $D(v)$ in Equation (1) shows how many nodes are interfered with node v in the IG. For a heterogeneous register architecture, it is in fact insufficient to show the interference relationship between two nodes since different register classes may be bound to the nodes. Two more factors must be considered here. First, the more registers a node has in its register class, the more number of the interfering nodes it has in the IG. Second, even when there is an edge between two nodes in the IG, there might be no actual interference between their live ranges if their register classes are disjoint. To reflect into the spill metric this interference relationship between nodes bound to different register classes, we devise a new interference degree $I(v)$:

$$I(v) = \varepsilon(\phi_v) \times \sum_{n \in adj(v)} f(v, n)$$

where *adj*(v) is a set of the neighboring nodes of v. For the register classes ϕ_v and ϕ_u, of two nodes v and u, we have $f(v,u) = 1$ if $\phi_v \cap \phi_u \neq \emptyset$, and

$f(v, u) = 0$ otherwise. $I(v)$ represents how many neighboring nodes are actually interfered with node v. Using $I(v)$, we propose a new spill metric $S(v)$:

$$S(v) = C(v)/I(v) \tag{2}$$

In Section 5, we will experimentally demonstrate that this new metric $S(v)$ improves the coloring of the IG, and also reduces the spills considerably. In the example of Figure 9, the original optimistic coalescing scheme could not efficiently break the tie between the spill metrics for nodes b and f when their spill costs $C(\text{b})$ and $C(\text{f})$ are identical. So in the experiment, it colors b first after live range splitting, resulting in an unnecessary spill. However, with our modified coalescing scheme with $S(v)$, we have $I(\text{b}) = 3$ and $I(\text{f}) = 1$, and consequently we have $S(\text{b}) = C(\text{b})/3$ and $S(\text{f}) = C(\text{f})$. Thus, even for $C(\text{b}) = C(\text{f}) > 0$, we have $S(\text{b}) < S(\text{f})$, which places node f in the higher priority for coloring, preventing the spill as the result.

4.3 Strategy for Coloring as Many Split Nodes as Possible

Instead of spilling a coalesced node, optimistic coalescing splits it into several "ingredient" nodes, trying to color as many nodes as possible among them. We call these ingredient nodes as primitive nodes, as denoted in [Park et. al. 2004]. If some nodes among the primitive nodes cannot be colored, they are immediately spilled. However, among the colorable ones, the node with the maximum spill metric can be immediately colored, as we explained in Section 3. For other colorable primitive nodes, we delay the decision of coloring them after all the nodes below the coalesced node in the stack are colored. In the original optimistic coalescing scheme, we just hope that as many nodes as possible could be colored then. In order to color the maximize number of the primitive nodes, we immediately color all the nodes whose register classes are as the same as that of the original coalesced node. As shown in Figure 11, when the coalesced node e is split into f and g, f is spilled immediately, but g is colorable. We delay coloring g until all the nodes in the stack like d are colored. In this case, if unfortunately r0 is assigned to d like in Figure 11(e), g cannot be colored. As the register class of g is the same as that of the original coalesced node e, we can assign a register to g in advance like in Figure 11(d). The colors (registers) assigned to these nodes does not ruin the coloring of other nodes left in the stack. It is due to the fact that the register class of the coalesced node is already considered when the other nodes below in the stack are pruned, and the register classes of those primitive nodes being immediately colored are the same as the one of the coalesced node. This coloring heuristic increases the number of the primitive nodes being immediately colored, which shows better register allocation with the decreased number of spills.

4.4 Copy Sifting

As we explained, optimistic coalescing takes the aggressive way of coalescing, where all the copy related nodes are merged before simplification. It tries to alleviate the negative impact of coalescing by splitting the coalesced nodes

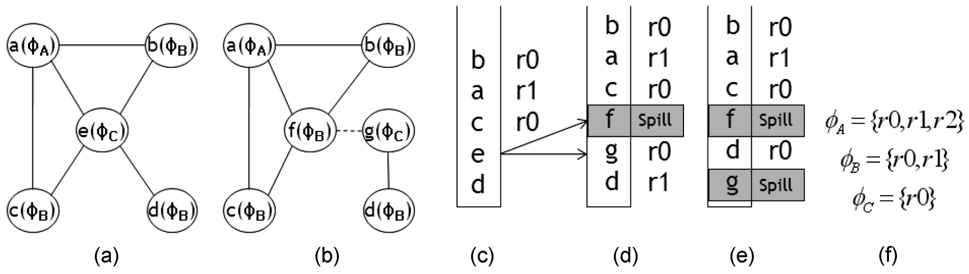


Fig. 11. An example of coloring as many split nodes as possible.

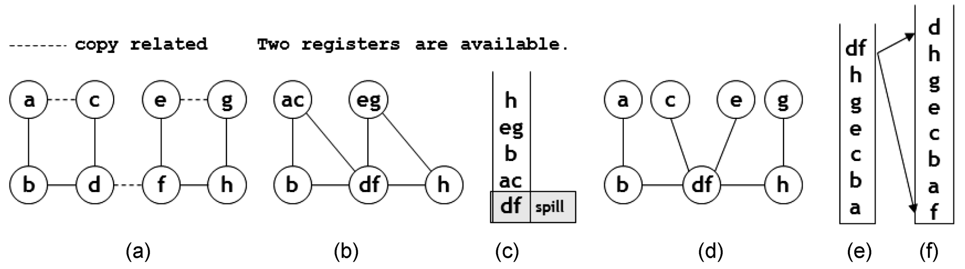


Fig. 12. Negative impacts of the coalesced neighboring nodes.

when it is chosen as spill. But there are some cases where we cannot avoid the spill of the coalesced node or all (some) of its primitive nodes if the coalesced node has the neighboring nodes which are already coalesced.

Figure 12(a) shows an IG where two registers are available and all the nodes have the same register class composed of these two registers. In this example, all the copy related nodes are aggressively coalesced as optimistic coalescing does. Figure 12(b) shows the result, where all the nodes are the significant nodes. If we assume that the spill costs of all nodes are the same, the node **df** is firstly chosen as a spill candidate due to its largest degree. After the coalesced node **df** is pruned, all the rest nodes can be pruned like in the stack of Figure 12(c). After simplification, all other nodes are colored except the node **df**. Instead of spilling it, optimistic coalescing splits it into the primitive nodes **d** and **f**, trying to color them all. Unfortunately, the coalesced nodes **ac** and **eg** adjacent to **df** are already colored, so we have to spill **d** and **f** like in Figure 12(c). We can avoid this if we do not coalesce some of the coalesced nodes. For example, if we coalesce only node **d** and **f**, not others, we get the IG of Figure 12(d). In this IG, all the nodes are pruned like the nodes of Figure 12(e).[3] They are colored following the order of popping nodes from the stack. Due to splitting the coalesced node **df,** we can color all the nodes without spills like in Figure 12(f). Likewise, we can avoid the unnecessary spills of the coalesced nodes or all (some) of its primitive nodes by sifting the copies when we apply the aggressive coalescing strategy. We call this technique copy sifting.

[3]There are other ways of pruning (simplification) in the IG of Figure 12(d). In any case, they do not produce spills.

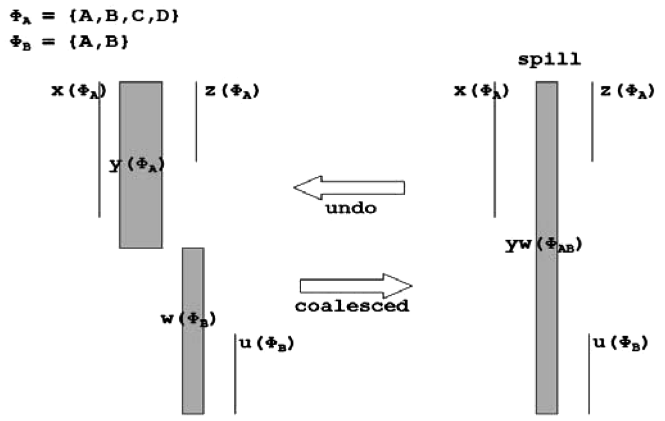


Fig. 13. The live range to be spilled after coalescing.

What is important in copy sifting is how to 'sift' copies. Once a copy is chosen not to be coalescible, it cannot be eliminated later like the copies (**a-c** and **e-g**) in Figure 12(d). Therefore, the more copies are chosen as not coalescible in copy sifting, the more number of redundant copies are left, resulting in the degradation of the code quality. On the other hand, the less copies are sifted, the more spills may be produced. This tradeoff makes copy sifting an interesting problem. We found that several things should be considered for maximizing the code quality in copy sifting. First, in the HRAs, we can use the similarity between register classes and the register class influence $\varepsilon(\phi_v)$ as the criterion of sifting. Figure 13 shows an example, where the vertical lines and the bars represent the live ranges of variables, and each variable has a register class ϕ_A or ϕ_B. In Figure 13, two copy related variables **y** and **w** have different register classes $\phi_y(=\phi_A)$ and $\phi_w(=\phi_B)$. The coalesced node **yw** has the register class $\phi_{yw}(=\phi_{AB})$ which is the intersection of those two register classes ϕ_A and ϕ_B. Now, the coalesced node **yw** has a long live range. If we assume that it becomes an actual spill later, **yw** is split into two primitive nodes **y** and **w**. Whether two primitive nodes are colored or not, it is not necessary to coalesce two nodes y and w aggressively because the coalesced nodes **yw** is undo-coalesced in live range splitting. In this case, two variables with different register classes ϕ_A and ϕ_B are coalesced into the one with a register class ϕ_{AB} whose size is smaller than that of the original register class ϕ_y. If the register classes of two coalescible nodes are very similar and the number of the common registers in both register classes is large, it is good to coalesce them aggressively. If not, coalescing them might be useless due to live range splitting, and worse, still produce additional spills like in Figure 12(b). As noted in the example of Figure 13, two different register classes ϕ_y and ϕ_w makes the coalesced nodes **yw** split later. If the register class ϕ_y is the same as the small-sized register class $\phi_w(=\phi_B)$, the coalesced node **yw** will be split for coloring which makes the aggressive coalescing strategy useless, too. However, if the register class ϕ_w is the same as $\phi_y(=\phi_A)$, the size of the register class $\phi_{yw}(=$ {A, B, C, D}$)$ is 4. In this case, the coalesced node **yw** can be colored and the aggressive coalescing is not nullified by live range splitting.

Likewise, in HRAs, unnecessary aggressive coalescing might be avoided when only the copy related nodes with 'similar' and 'large-sized' register classes are coalesced. How much the size of register class ϕ_v is can be represented using the register class influence $\varepsilon(\phi_v)$, as explained in Section 4.2. In order to use the similarity between the register classes as a criterion of copy sifting, we define the similarity between two register classes here.

Definition 4. The similarity $\theta(\phi_v, \phi_u)$ is defined between the register classes ϕ_v and ϕ_u, of two nodes v and u, as follows:

$$\theta(\phi_v, \phi_u) = \frac{1}{2} \times \left(\frac{|\phi_u \cap \phi_v|}{|\phi_u|} + \frac{(|\phi_u \cap \phi_v|)}{|\phi_v|} \right)$$

If two register class ϕ_v and ϕ_u are the same, $\theta(\phi_v, \phi_u) = 1$. On the other extreme, if ϕ_v and ϕ_u have no common registers, $\theta(\phi_v, \phi_u) = 0$.

The other factor that should be considered in copy sifting is how much benefit we get from coalescing is. In order to estimate the benefit from coalescing, both the positive impact of coalescing and the negative impact should be considered. The positive impact of coalescing decreases the number of edges in the IG. It may result in less number of spills because the less number of edges might be less squeeze of each node in the IG, as explained in Section 2.3 and Section 3. So the more the positive impact is, the more benefit from the coalescing is. On the other hand, the negative impact of coalescing increases the degree of each coalesced node in the IG, increasing the possibility of producing more spills. The more the negative impact is, the less benefit from the coalescing is. As explained in Section 3, the aggressive coalescing strategy in optimistic coalescing can fully enjoy the benefit from the positive impact of coalescing, and live range splitting can alleviate the negative impact of coalescing. However, as shown in the example of Figure 12, we still have more chances to suppress the negative impact of coalescing by copy sifting. In order to maximize the benefit from coalescing, it is necessary to measure it. Unfortunately, it is hard to estimate the negative impact of coalescing before coalescing. Although the negative impact increases the degree of each edge in the IG as stated above, the increased number of edges after coalescing does not always produce a spill due to live range splitting. The example is the coalesced node **df** in Figure 12(d). On the other hand, it is easy and accurate to estimate the positive impact of coalescing. We just calculate the number of the common neighboring nodes when we coalesce two copy related nodes in the IG. The more common neighboring nodes they have, the more positive impact of coalescing they have. Therefore, we define the positive impact of coalescing ρ_{vu} as follows when we coalesce two copy related nodes v and u:

$$\rho_{vu} = \frac{|adj(v) \cap adj(u)|}{|adj(v) + adj(u)|}$$

where *adj(u)* is the set of the neighboring nodes of u in the IG. If two nodes v and u have only common neighbors, ρ_{vu} is 0.5. If they have no common neighbors, ρ_{vu} is 0. Now we can determine whether it is beneficial to coalesce a copy or not by using the above considerations before coalescing, and sift it.

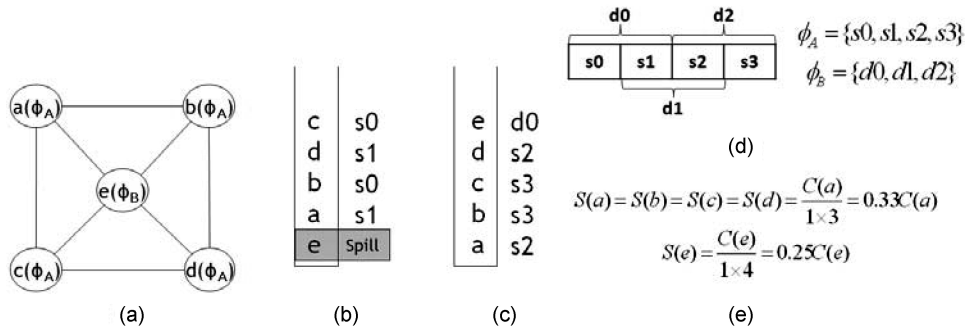

a(ϕA)
b(ϕA)
e(ϕB)
c(ϕA)
d(ϕA)
(a)
c s0
d s1
b s0
a s1
e Spill
(b)
e d0
d s2
c s3
b s3
a s2
(c)
d0 d2
s0 s1 s2 s3
d1
$\phi_A = \{s0, s1, s2, s3\}$
$\phi_B = \{d0, d1, d2\}$
(d)
$S(a) = S(b) = S(c) = S(d) = \frac{C(a)}{1 \times 3} = 0.33C(a)$
$S(e) = \frac{C(e)}{1 \times 4} = 0.25C(e)$
(e)

Fig. 14. A example of selecting potential spill with/without considering the register alias.

Definition 5. Sifting criterion τ: When we coalesce two copy related nodes u and v, we determine whether it is beneficial or not by the sifting criterion τ_{uv}

$$\tau_{vu} = \varepsilon(\phi_v \cap \phi_u) \cdot \theta(\phi_v, \phi_u) \cdot (1 + \rho_{vu})$$

We do not aggressively coalesce v and u if $\tau_{vu} < \tau_T$, where τ_T is a threshold for sifting.

4.5 Impact of Register Alias on Our New Spill Metric

In Section 4.2, we propose a new spill metric $S(v)$. Using $S(v)$, we select a potential spill candidate among the significant nodes in the IG and decide the immediate colorable node in live range splitting. In $S(v)$, we consider the effect of the register class of each node on its neighboring nodes by the register class influence ε, obtained from RCT. This register class influence is the inverse of the depth in RCT. As the depth represents the inclusion relationship between the register classes in target architecture, it is the same as the number of registers in each register class is considered. We discover that not only the number of registers in each register class but also the register aliasing explained in Section 2.2, is important. Figure 14 shows an example.

In this example, there are four physical registers s0, s1, s2, and s3. And the registers d0, d1, and d2 have the alias relationship to the registers s0, s1, s2, and s3, as shown in Figure 14(d). Each node in the IG of Figure 14(a) has a register class ϕ_A or ϕ_B. As all the nodes in the IG are significant nodes, we select a potential spill by comparing the new spill metric of each node. We assume that the spill costs of all nodes are the same and the register class influence of each register class is 1. Figure 14(e) shows the new spill metric of each node. The spill metric of node **e** is the smallest, so we select it as a potential spill. Then other nodes can be pruned into the stack of Figure 14(b). When we try to color each node from the stack, node **e** is spilled. As a large-sized register in ϕ_B includes two small-sized registers in ϕ_A due to register aliasing, it has less freedom to color **e** later by selecting it as a potential spill. Therefore if we select a node except **e** as potential spill (e.g., node **a** in Figure 14(c)), we can color all the nodes in the IG without spills. Generally, the register class with large-sized registers aliased to other small-sized registers (e.g., double-word registers like d0, d1, or d2 of the example) can further prevent the coloring of the neighboring nodes having the register class with the small-sized registers. If we consider the effect of it in the new spill metric, we can avoid unnecessary spills like in Figure 14(c). We define *register size influence* ω, which represents the effect of the size of registers in each register class. All the registers in each register class has the same sized registers. We modify the new spill metric Equation (2) by adding the ω as follows:

$$S(v) = \omega(\phi_v) \cdot C(v)/I(v) \tag{3}$$

where $\omega = 1$ for the register class with single word registers. In the register class whose register size is larger than a single word size of target architecture, ω is the value that is obtained from dividing the size of a register by the word size. For the register class whose register size is smaller than that the size of a single word, $\omega = 1$.

5. EXPERIMENTS

In our experiment, we compare our modified optimistic coalescing scheme with two different schemes: that is, one is the existing scheme based on iterated coalescing in a related work, and the other is the original optimistic coalescing scheme. Probably the work most closely related to ours is the one conducted by [Smith et al. 2004]. As stated earlier, ours and theirs basically use almost the same register allocation framework for heterogeneous register architectures, but in their framework, they adopt a special scheme based on iterated coalescing for heterogeneous registers while we adopt a modified optimistic coalescing scheme described in Section 4. In our experiment, we try to compare the performance of our scheme and theirs. For fair comparison, we implement both schemes in our register allocation framework. Also, for the other comparison, we test how much our new spill metric $S(v)$ improves the performance of optimistic coalescing with the original spill metric $M(v)$ for allocating heterogeneous registers to node v in the IG. So, in this section, we will report the performance results of two optimistic coalescing schemes each with $M(v)$ and $S(v)$. Also we will show what impact the traditional optimizations such as copy-propagation and common-subexpression elimination has on our register allocation algorithm and three coalescing schemes.

5.1 Experimental Environment

Our experiments have been performed on DSP563xx (refer to Figure 1) from FreeScale, which is a popular low end DSP with typical heterogeneous register architecture. We compile the benchmark codes using our retargetable compiler platform SoarGen. As shown in Figure 15, the machine dependent modules of our compiler are automatically generated from architecture description language (ADL). So we describe the ISA of DSP563xx using our ADL (SoarDL), and generate its compiler. Our compiler uses GNU gcc 3.3 c-compiler as front end. The front end generates the virtual assembly code based on a virtual machine that we assume. Its ISA is very simple and similar

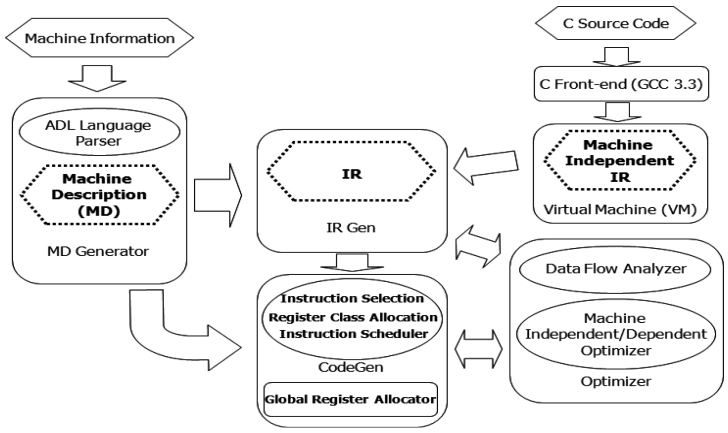


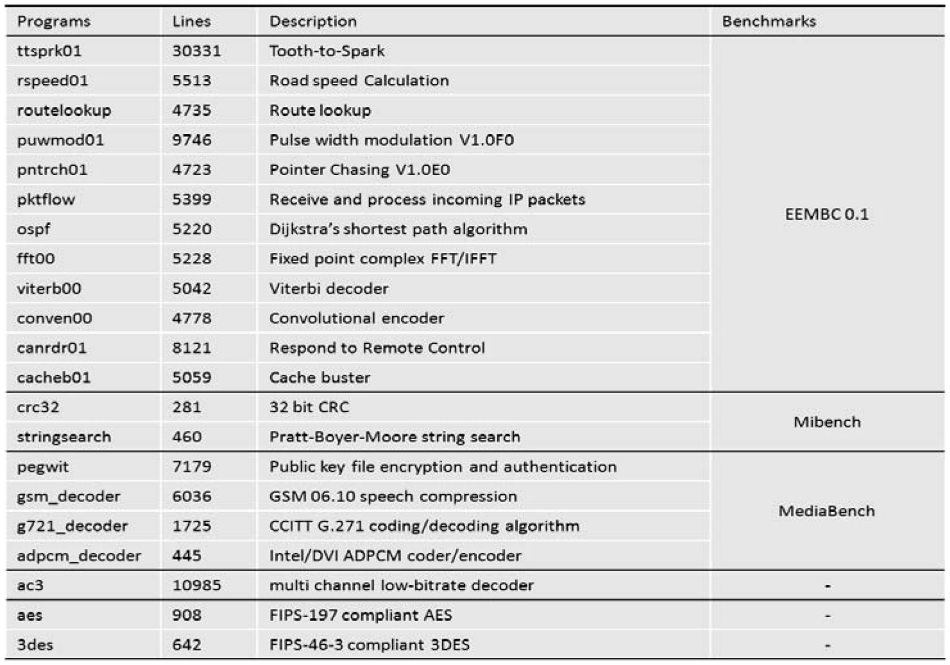
Fig. 15. The structure of SoarGen retargetable compiler platform.

with a typical load/store based RISC architecture. The back end of our compiler takes the virtual assembly codes as input and generates the target machine code. Due to gcc 3.3 c-compiler, we can apply many traditional optimization techniques such as dead code elimination, copy propagation and common subexpression elimination during code generation. Our instruction selector does partially-coupled code generation where the register classes in DSP563xx are assigned to all the operands of the instructions before register allocation [Ahn et al. 2007].

We ran our compiler on 3.2Hz Pentium IV with 2GB RAM. Our benchmarks include twenty one nontrivial integer DSP core applications shown in Figure 16. Benchmarks are of various sizes from MiBench to MediaBench [Lee et. al. 1997] and EEMBC 0.1. The smallest ones are crc32 and stringsearch from MiBench, which are the kernel codes frequently used in many DSP applications. The larger ones are ac3, pegwit, and gsm_decoder, which are all typical applications in the DSP domain. As the expensive floating-point calculation should be avoided in embedded system, we try to choose the benchmark codes consisting of only integer calculation. For some benchmark codes like ac3, we manually convert the floating-point calculation into integer calculation. Instead, in order to show you what each benchmark code is, we briefly explain the core feature of each benchmark code in Figure 16.

5.2 Comparing Three Coalescing Techniques

Figure 17 shows the features of the IR code right before register allocation. The ratio of copy instructions is very high. It is about 55% of the IR code. This is in fact not the case only for our compiler; other compilers such as the one in Zivojnovic [1996] also report a similar ratio of copy instructions during their code generation for heterogeneous register architectures. Actually, this high ratio of copy instructions gives us the chance of optimization for eliminating



Programs	Lines	Description	Benchmarks
ttsprk01	30331	Tooth-to-Spark	EEMBC 0.1
rspeed01	5513	Road speed Calculation	
routelookup	4735	Route lookup	
puwmod01	9746	Pulse width modulation V1.0F0	
pntrch01	4723	Pointer Chasing V1.0E0	
pktflow	5399	Receive and process incoming IP packets	
ospf	5220	Dijkstra's shortest path algorithm	
fft00	5228	Fixed point complex FFT/IFFT	
viterb00	5042	Viterbi decoder	
conven00	4778	Convolutional encoder	
canrdr01	8121	Respond to Remote Control	
cacheb01	5059	Cache buster	
crc32	281	32 bit CRC	Mibench
stringsearch	460	Pratt-Boyer-Moore string search	
pegwit	7179	Public key file encryption and authentication	MediaBench
gsm_decoder	6036	GSM 06.10 speech compression	
g721_decoder	1725	CCITT G.271 coding/decoding algorithm	
adpcm_decoder	445	Intel/DVI ADPCM coder/encoder	
ac3	10985	multi channel low-bitrate decoder	-
aes	908	FIPS-197 compliant AES	-
3des	642	FIPS-46-3 compliant 3DES	-

Fig. 16. Benchmark program.[4]

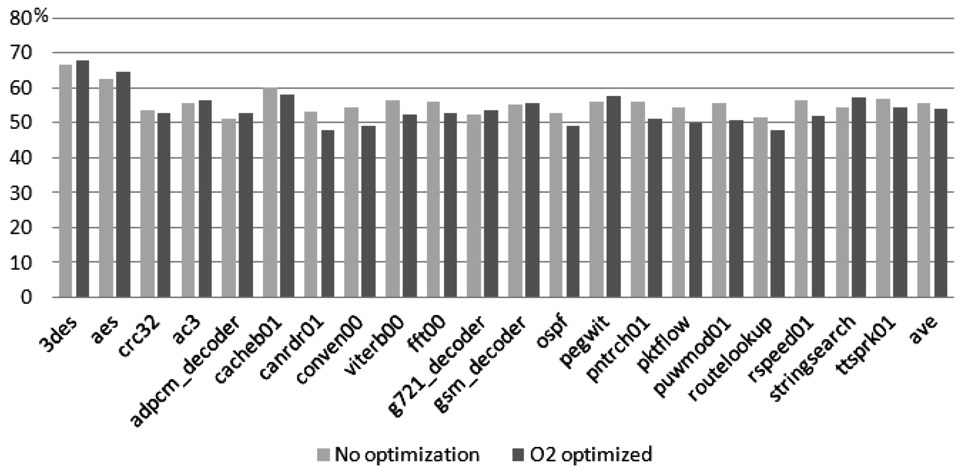
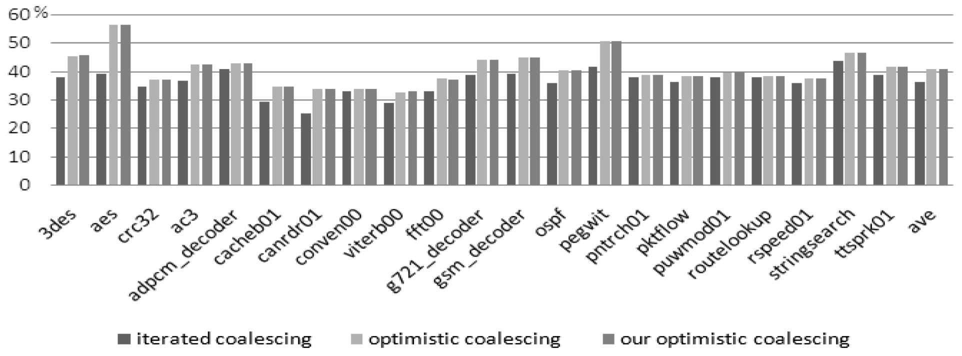
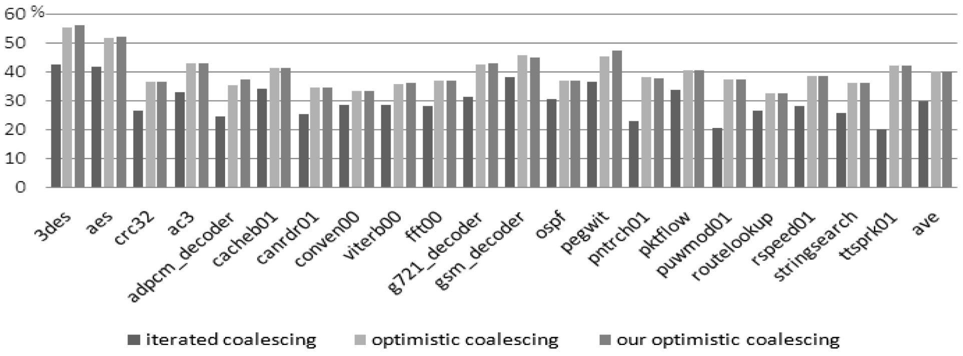


Fig. 17. Ratio of the copies in IR code right before register allocation.[5]

[4]We exclude some benchmarks codes including all small benchmarks from DSPStone [Zivojnovic et al. 2004] used in our previous work. Instead, we do our experiment for larger benchmarks from various benchmarks such as EEMBC, Mibench, and MediaBench.
[5]In all figures from now on, the last bar named 'ave' represents the average value, not benchmark result.



(a) No Optimizations are applied



(b) O2 Optimizations are applied

Fig. 18. Ratio of removed copies compared to those before coalescing: (a) No optimization is applied, and (b) O2 optimization is applied.

copies by using the coalescing techniques. When we apply O2 optimizations,[6] the ratio of copy instructions is decreased a little bit in most of the benchmark codes. It is because the traditional optimizations for minimizing the code size such as copy propagation are applied. After those optimizations are applied in the virtual assembly code by our front end, our compiler produces additional copy instructions in the instruction selection stage, which makes the ratio of copy instructions still high [Ahn et al. 2007].

In Figure 18, we compare how many copies can be removed in the three coalescing schemes. As you can see, the iterated coalescing scheme in Smith et al. [2004] cannot coalesce many copy related nodes, as compared to both optimistic coalescing schemes. For example, in ttsprk01, iterated coalescing eliminates only 20.1% of copies while optimistic coalescing eliminates up to 42.0% of them when O2 optimizations are applied. (Our coalescing scheme eliminates 42.3% of them.) In all the benchmark codes, optimistic coalescing merges much more

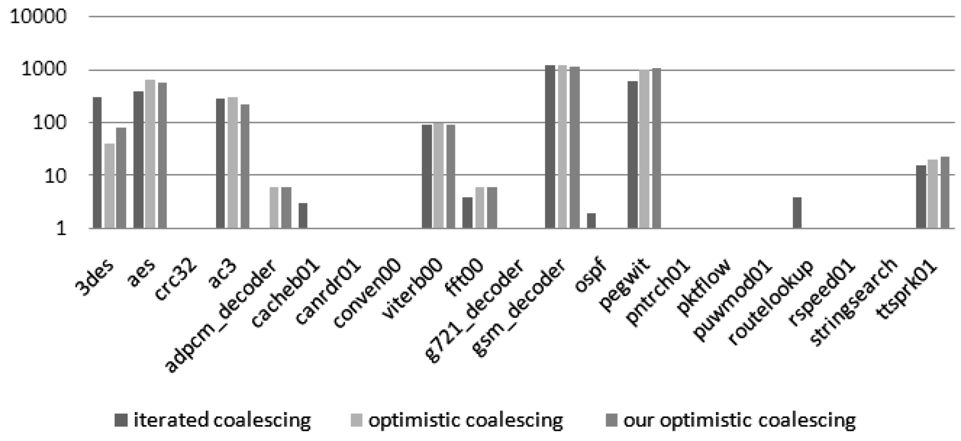
[6]Our front-end gcc 3.3 c-compiler does O2 optimizations. O2 optimizations include many traditional optimizations like copy propagation, and common subexpression elimination. It is for minimizing the code size.

copies than iterated coalescing. The reason is that, as explained earlier, iterated coalescing takes too conservative coalescing strategy, thus giving up the chance of coalescing too early. When no optimizations are applied, the ratio of the eliminated copies is higher in the iterated coalescing scheme. It is because the live ranges with no optimizations applied have less conflict to others, compared to the one with O2 optimizations applied.[7] The live ranges with less conflict produce the less number of the interfering edges in the interference graph. When a live range has less number of conflicts like this, the coalesced nodes may have less number of conflicts, which can increase the number of removed copies without using the aggressive coalescing scheme like optimistic coalescing. In this case, the conservative coalescing strategy can show not so inferior performance. Iterated coalescing eliminates up to 36.2% of the copies, which is close to that of both optimistic coalescing (40.8%). However, the gap between the ratios of the eliminated copies is larger when O2 optimizations are applied (iterated coalescing: 29.9%, optimistic coalescing: 39.9%, our optimistic coalescing: 40.2%). In any case, being compared with the original optimistic scheme, our modified scheme eliminates more copies.

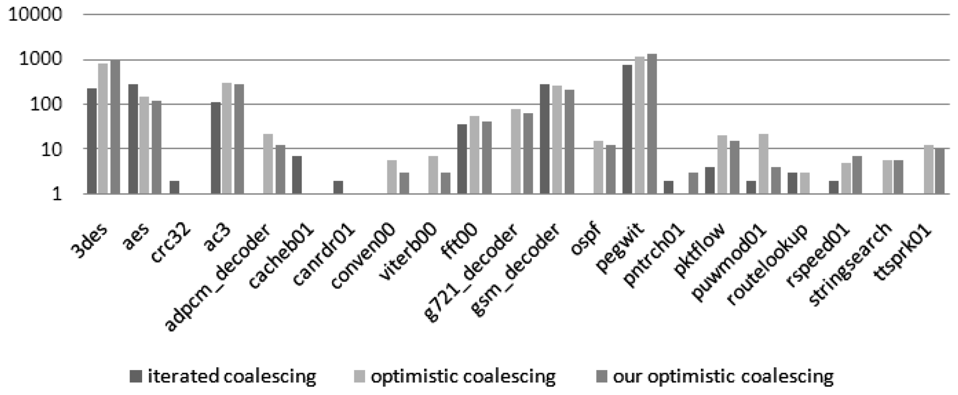
Figure 19 illustrates how many spills are produced by the three coalescing schemes. As stated earlier, the live ranges of the variables in the codes with no optimizations applied have relatively less conflicts with other live ranges, so many of the spills are determined by the way of register allocation and the coalescing schemes. Therefore, in some benchmark codes with no optimizations applied, (3des, cacheb01, ospf, gsm_decoder, and routelookup) both optimistic coalescing schemes insert smaller number of spills than the iterated coalescing scheme. This is mainly because optimistic coalescing minimizes the negative impact of coalescing via live range splitting. But, in other codes with no optimizations applied, like aes, adpcm_decoder, viterb00, pegwit, and ttsprk01, both optimistic coalescing techniques insert more spills than the iterated coalescing technique. It is because many of the nodes coalesced by optimistic coalescing are spilled despite of live range splitting. With O2 optimizations applied, where each live range conflicts with more other live ranges, this is the case.

Luckily, in our optimistic coalescing, the new spill metric and the coloring heuristic help us to choose the better nodes for selecting potential spill and coloring all the split nodes during live range splitting, especially with O2 optimizations applied. Therefore, as displayed in Figure 19, our modified coalescing scheme produces fewer spills than the original optimistic coalescing schemes in most benchmark codes. 9.0% of spills are reduced more, in the codes with O2 optimizations applied, but there are some benchmark codes like 3des, and pegwit, where our scheme produces more spills. As previously explain in Section 4.4, it is due to the aggressive way of coalescing in the optimistic coalescing schemes. As live range splitting considers only one node at a time, if the coalesced node

[7]How much conflict a live range has depends on the kinds of the applied optimizations. But in this context, we wanted to explain the difference between O0 optimizations and O2 optimizations in our experimental environment. (We think that this may be true in many other compilers) We examine how many conflicts a live range has in each optimization. This data is collected from the first live range analysis of the register allocation in each optimization. In O0 optimizations, it has 5.79 conflicts on average. In O2 optimizations, it has 8.38 conflicts on average.



(a) No Optimizations are applied.

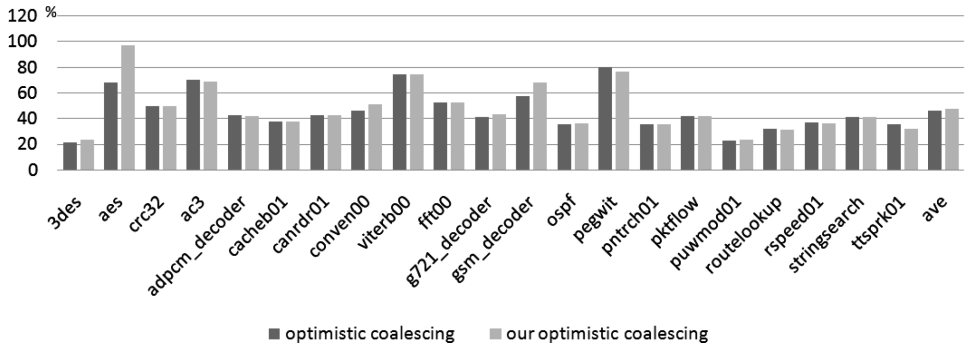


(b) O2 Optimizations are applied.

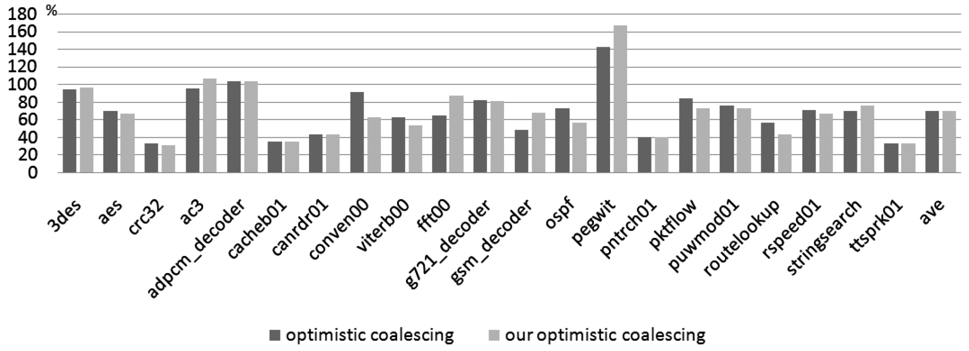
Fig. 19. The number of spills produced by three coalescing schemes.

has many other coalesced nodes in neighbors, it increases the number of the significant neighboring nodes, producing inevitable spills despite live range splitting.

Unfortunately, the improvement in our coalescing scheme comes at the extra cost of compilation time and memory requirement. In our scheme, we need to maintain the RCRT and the RCT for the coalescing, which can be ignored in comparison with the memory size needed for the whole register allocation algorithm. Figure 20 shows the register allocation time of the original optimistic scheme and that of our optimistic scheme. In the figure, both times are normalized to that of iterated coalescing. Note that in almost all benchmarks, the register allocator with both the optimistic schemes runs faster than the one with the iterated scheme. This result is actually not surprising because iterated coalescing needs to repeat multiple times the simplification, coalescing and freeze phases, as shown in Figure 5. Our optimistic coalescing technique generally takes more time because it should do additional jobs described in



(a) No Optimizations are applied

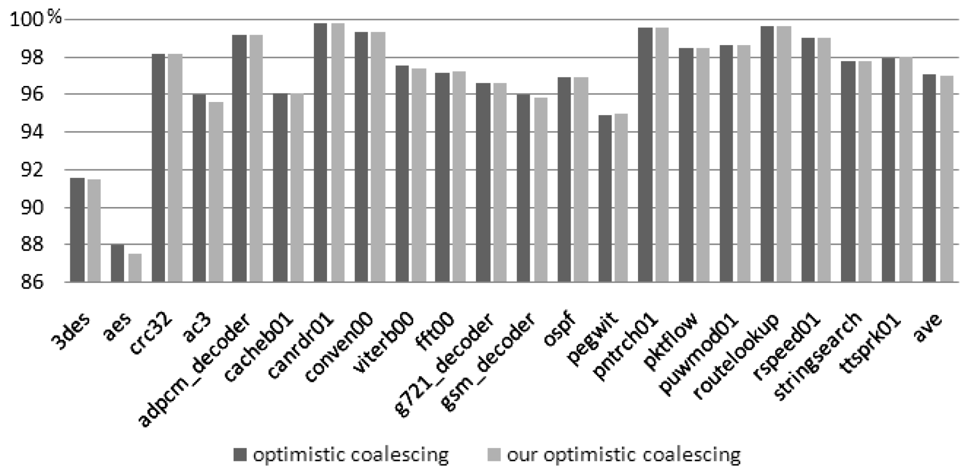


(b) O2 Optimizations are applied

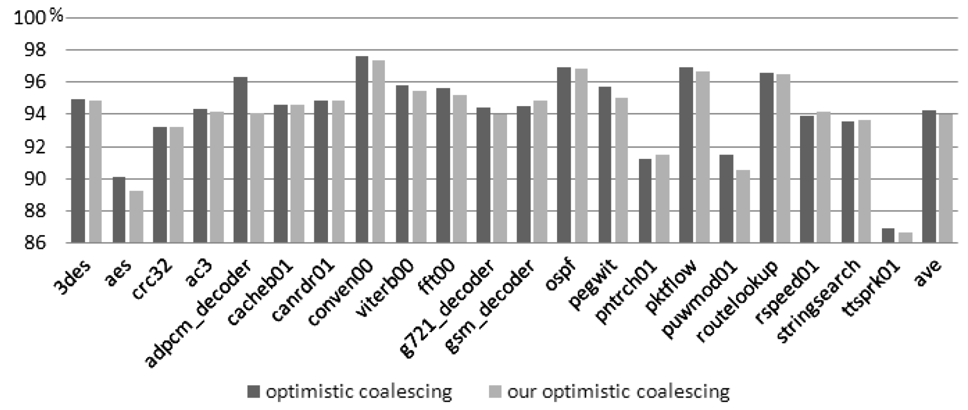
Fig. 20. Ratio of compile time for register allocation with two optimistic coalescing schemes normalized to that with iterated coalescing.

Section 4, but sometimes in our optimistic coalescing technique, a successful allocation or coalescing in the a round of the register allocation does not require the next round of the register allocation, where it takes less time than the original optimistic coalescing technique.

Although there are some benchmark codes where optimistic coalescing produces more spills (especially the ones with O2 optimizations applied), we observe that optimistic coalescing generally outperforms iterated coalescing in various aspects because optimistic coalescing can eliminate more copy instructions (4.5% more on average with no optimizations applied, 10.0% with O2 optimizations applied) and produces less spills (15% less on average with no optimizations applied), as shown in Figure 18 and Figure 19. In Figure 21, we show the final code sizes of both optimistic coalescing schemes normalized to those of the iterated coalescing scheme. The original optimistic scheme reduces the code size by up to 13.0% (on average 5.7%), as compared to iterated coalescing when O2 optimizations are applied (on average 3.0% with no optimizations applied). Our optimistic scheme further reduces the code size by 1.0% on average in O2 optimizations. This code size reduction is small in our optimistic coalescing scheme, but our coalescing scheme produces much less



(a) No optimizations are applied.



(b) O2 Optimizations are applied

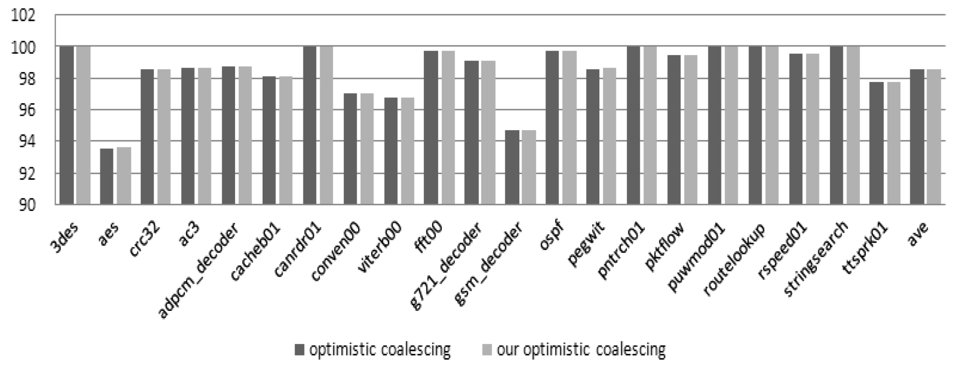
Fig. 21. Ratio of total code size compared to iterated coalescing.

spills compared to the original optimistic coalescing scheme (9% less on average with O2 optimizations applied). This is thanks to the combined effects of aggressive coalescing and live range splitting using our new spill metric $S(v)$ and the coloring heuristic, mitigating the negative effect of aggressive coalescing. The runtime result in Figure 22 shows that both optimistic coalescing schemes generate fast code than the iterated coalescing scheme. (1.4% decrease with no optimizations applied, and 5.5% with O2 optimizations applied). However, there is no big difference between the runtime performances of the original coalescing scheme and our optimistic coalescing scheme. In order to get more runtime performance increase in our optimistic coalescing scheme, we reduce more spills by applying copy sifting. This result is in Section 5.4. In the next section, we will examine the negative impact of optimistic coalescing.

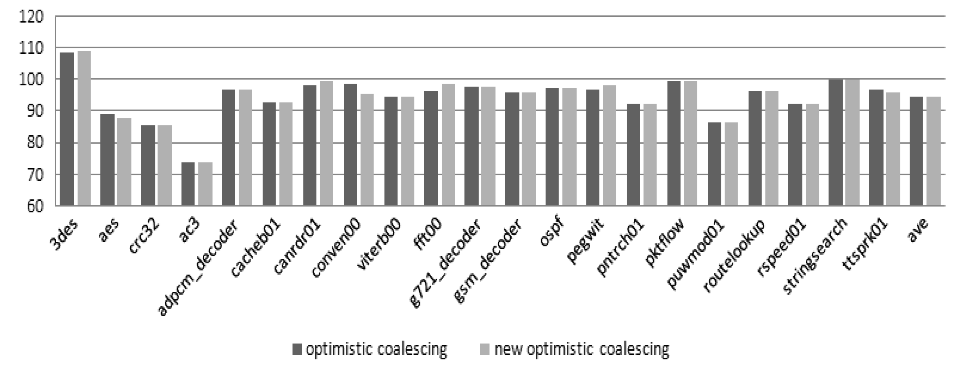
5.3 Negative Impact of Coalescing In Two Optimistic Coalescing Schemes

As explained in Section 3, the negative impact of optimistic coalescing increases the degrees of the coalesced nodes in the IG, producing more spills. We estimate its effect by measuring how many nodes that are coalesced despite violating the conservative coalescing strategy used in iterated coalescing are actually spilled. As explained in Section 4.3, several primitive nodes are included in a coalesced node. A coalesced node, which is coalesced despite violating the conservative coalescing strategy, is called a violating coalesced node. Figure 23 shows the ratio of the number of actually spilled nodes to the number of nodes in the violating coalesced nodes. In the original optimistic coalescing scheme, an average of 13.9% of nodes is actually spilled when O2 optimizations are applied. This rate further drops down to 13.1% on average in our optimistic coalescing with the help of the combined effects of aggressive coalescing and live range splitting using our new spill metric $S(v)$ and the coloring heuristic. As the low ratio means the smaller number of spills, the benefit from the aggressive coalescing strategy gets bigger due to the large positive impact of coalescing, which the conservative coalescing strategy used in iterated coalescing would miss even if they can be colored successfully after being coalesced. When no optimizations are applied, this rate is 6.2% on average.

As explained in Section 3, live range splitting would help mitigating the negative impact of coalescing. We investigate this effect of the live range splitting of coalesced nodes in the original optimistic coalescing scheme and our optimistic coalescing scheme. Coalesced nodes are categorized into three according to the result of live range splitting: (1) All the primitive nodes are spilled. (2) All the primitive nodes are colored successfully. (3) Some primitive nodes are colored, whereas others are spilled. The nodes in (1) are the same as the coalesced node itself is spilled. Only in (2) and (3), optimistic coalescing can enjoy the goodness of live range splitting. Figure 24 shows the ratio of the number of successfully colored nodes (this includes the cases where some of the primitive nodes in the violating coalesced nodes are colored) to the number of all the nodes in violating coalesced nodes. When no optimizations are applied, this ratio is 89.5% in the original optimistic coalescing scheme. This is very high ratio compared to the result of [Park et. al. 2004]. In their work, this ratio is 17.1% in the original optimistic coalescing scheme. Our result means that almost all the primitive nodes in spilled coalesced nodes can be colored by undo-coalescing except 9.5% of the nodes. We think that it is due to the feature of the register classes of our target architecture and the register allocation algorithm. Each register class in our target architecture has small number of registers inside, as shown in Figure 2. The squeeze $\sigma_v(\varphi_u)$ of a node v caused by the neighboring nodes $u(\in U)$ is bound to $|\varphi_v \cap \varphi_u|$, where U is a set of the neighboring nodes with the same register class [Smith et al. 2004]. So although the degree of the neighboring nodes with the same register class is very high, the node can still have chances to color using the registers in the non-intersecting part of φ_v and φ_u. In Park et al. [2004], the target architecture (SPARC machine) is homogeneous register architecture; that is, its register class is one because they use a single large register file (32 general purpose registers). In this case, the high degree



(a) No optimizations are applied.



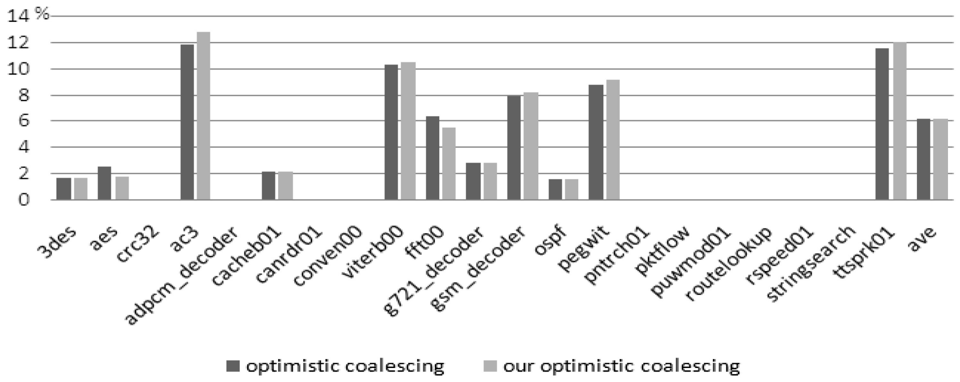
(b) O2 Optimizations are applied

Fig. 22. Ratio of runtime compared to iterated coalescing.

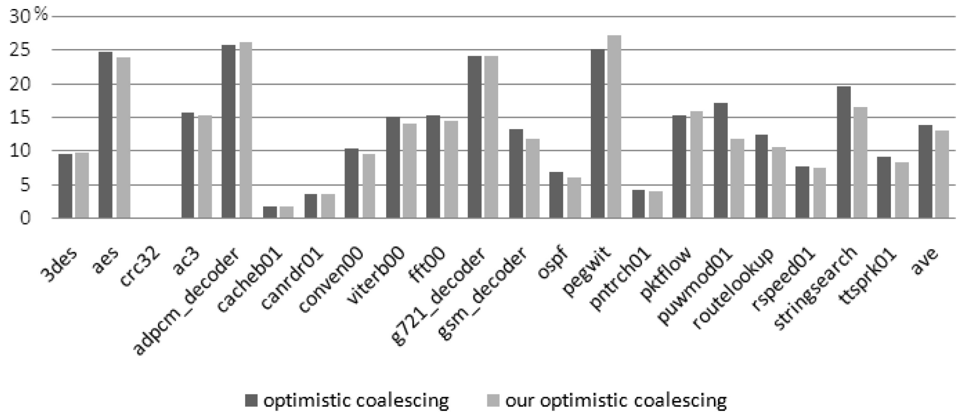
of a node directly makes the node spilled. That is why the successful coloring ratio is so high in our experiment, which is the evidence that the optimistic coalescing scheme shows good performance in HRAs. In other point of view, this high successful coloring ratio also informs us that in live range splitting, the original places where the eliminated copies exist is good to undo-coalesce because most of the split live ranges (i.e., primitive nodes in the coalesced node) are successfully colored. In our modified coalescing scheme, the ratio of the successfully colored nodes is 91.5% due to the new spill metric and the coloring heuristic. With O2 optimizations applied, these ratios go up to 95.8% and 96.1 each.

5.4 Further Decreasing Spills

As previously shown, our modified optimistic coalescing scheme reduces more spills in most benchmark codes compared to the original optimistic coalescing scheme, but in some benchmark codes such as pegwit and 3des, our modified optimistic coalescing scheme produces more spills. In order to reduce the number of spills in these cases, we introduce copy sifting described in Section 4.4.



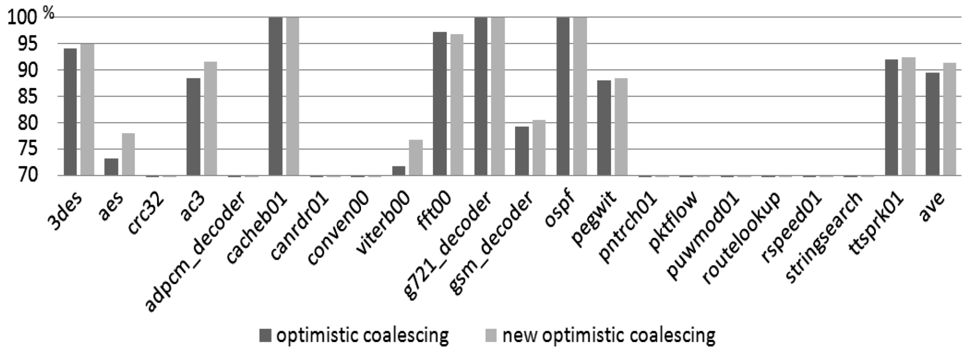
(a) No Optimizations are applied



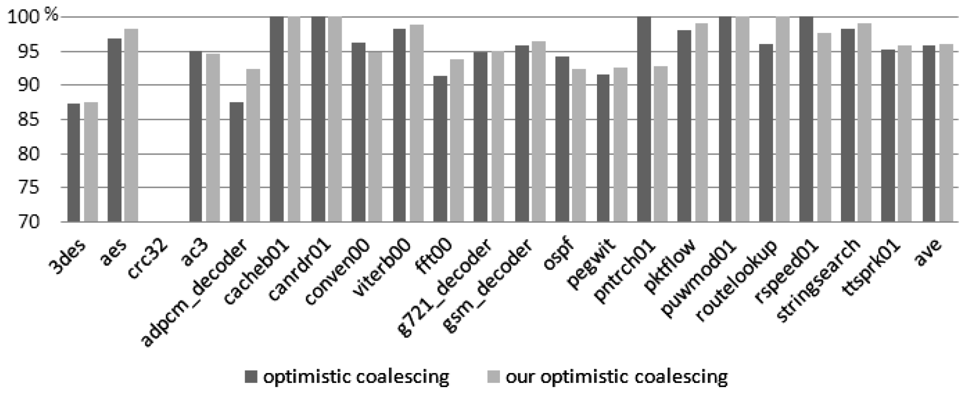
(b) O2 Optimizations are applied

Fig. 23. Ratio of spilled nodes in violating coalesced nodes.

Figure 25 and Figure 26 show the result. In these experiments, we apply copy sifting to our modified optimistic coalescing, and measure the change of the code quality in all benchmark codes, changing τ_T from 0.0 to 1.0 by 0.1. Our compiler does register allocation eleven times per benchmark codes, and select the best quality code, even though it takes eleven times more time. We summarize the code quality using three factors—the number of spills, the number of left copies after finishing register allocation, and the code size in Figure 25. All of them are normalized to the results in the case at threshold 0.0. The figure shows the average values of the results from all benchmark codes. The three lines in the figure show them. The upper one represents the change in the number of the left copies. The middle one represents the change of the code size. The lower one represents the change of the number of spills. As expected, the number of spills is decreased against the increase of τ_T. The number of spills is not changed from 0.0 to 0.5. It starts to be decreased from the threshold 0.6. At threshold 0.7, 70.9% of the spills are eliminated more. It continues to be decreased, but



(a) No optimizations are applied.



(b) O2 optimizations are applied.

Fig. 24. Successful coloring ratio of live range splitting in the original optimistic coalescing scheme and our optimistic coalescing scheme.

the amount of decrease is so small. Finally when we aggressively coalesce no copies (τ_T is 1.0), 78.8% of spills are eliminated more in our modified coalescing scheme. As explained above, we should pay for this large amount of reduction in spills at the cost of the losing chances of coalescing in copy sifting. As we decrease the spills, we lose more chances of coalescing, left more copies uncoalesced, resulting in the increase of the left copies and the code size. As shown in Figure 25, the number of the left copies and the code size are increased along with τ_T. From 0.0 to 0.5 of τ_T, they are not changed. At 0.6, the code size is increased by 0.3%. It is because the increase of the uncoalescible copies in copy sifting overwhelms the decrease in spills. At 0.7, the code size is increased by 1.3% more. The code size continues to be increased, resulting in 5.5% of increase at threshold 1.0. It means that the code size will be increased by 5.5% if we do not coalesced any copy related nodes in the IG at all. From the result in Figure 25, we conclude that we can get the best code quality at threshold 0.7, where the spills are decreased by 70.9%, whereas the code size is increased

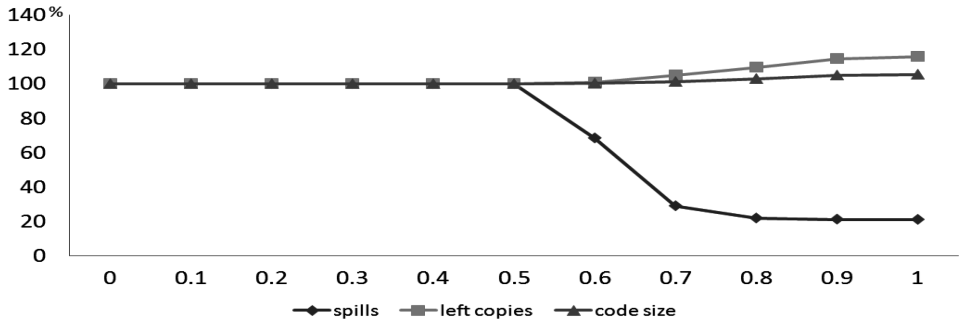


Fig. 25. The change of code quality with threshold from 0.0 to 1.0.

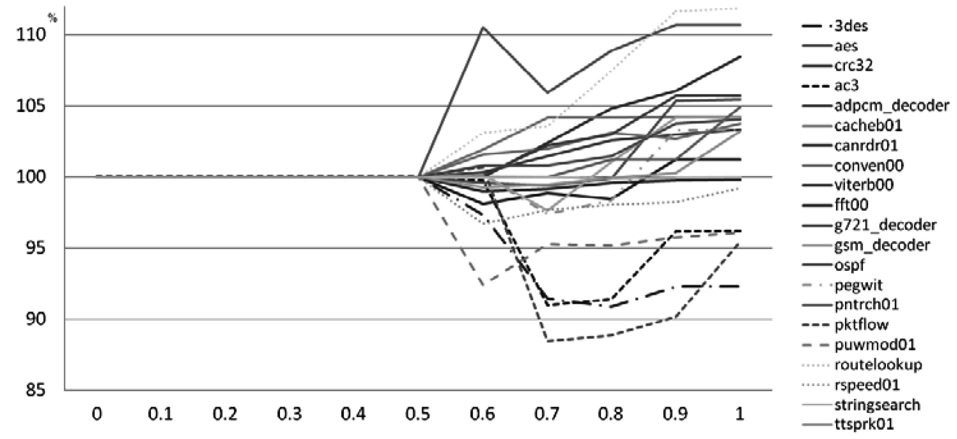


Fig. 26. The change of runtime with threshold from 0.0 to 1.0.

only by 1.3%. As stated in Section 5, our modified optimistic coalescing scheme decreases the code size by 1.0% more, comparing with the original optimistic coalescing scheme. So, if we apply copy sifting with the threshold value 0.7, our modified coalescing scheme can eliminate large amount of spills with almost the same code size as the original coalescing scheme.

Figure 26 shows the change of the runtime with threshold from 0.1 to 1.0 in all benchmark codes. Similar to the result in Figure 25, at the threshold 0.7, the runtime is decreased by 1.0% on average. However, in some benchmark codes like ac3, 3des, and pktflow, the runtime is decreased by about 10%, due to copy sifting. In these benchmark codes, copy sifting is very efficient because it reduces the runtime up to 10% with 1.0% code size increase.

Figure 27 show the decrease of spills after we apply Equation (3) to our modified optimistic coalescing. In each benchmark code, upper bar is the number of spills when we apply Equation (2) (without ω) and lower bar is the number of spills when we apply Equation (3) (with ω). As many double-word register accesses exists in the IR codes before register allocation, we focus on reducing spills in two benchmark codes-ttsprk01 and aes. In this experiment, we apply

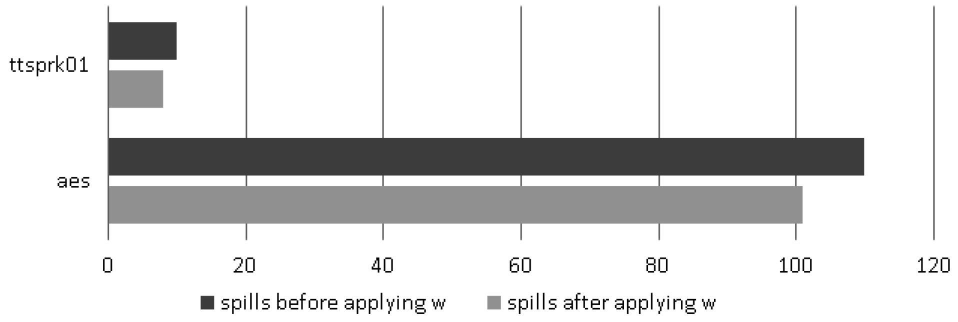


Fig. 27. The ratio of spills before/after applying register size influence ω to the new spill metric in Equation (2).

the new spill metric with ω when we break the tie where all the significant nodes including the ones with the register class having large-sized registers have the same spill metric value. In ttsprk01 and aes, we more reduce 2 and 10 of spills each. Although these reductions make the number of the removed copies from coalescing a little bit, their final code sizes have no big difference. In ttsprk01, the number of the removed copies is decreased from 993 to 992. In aes, it is decreased from 3,913 to 3,907, so their code sizes are not so changed. We think this spill metric is helpful to reduce the spills without the code size increase in many other benchmark codes where frequent double-word register accesses are required.

6. CONCLUSION

This article reports our recent efforts of building a compiler with a register allocation framework specifically designed to handle the heterogeneity of register architectures commonly encountered in many embedded processors. Since register coalescing is indispensable for the register allocator, we implement a coalescing algorithm in our register allocation framework. In order to coalesce as many copy instructions as possible, we resort to optimistic coalescing instead of iterated coalescing. Optimistic coalescing generates a better quality code than iterated coalescing. Unfortunately, the original optimistic scheme cannot handle heterogeneous registers effectively, thus producing unnecessary spills. To tackle this issue, we adapt the modified optimistic coalescing scheme into our register allocation framework. In this work, we cope with additional constraints that the heterogeneous register architectures requires to determine the register classes of coalesced nodes in the IG and to choose the immediately colorable node among all split nodes after live range splitting. Our experiments show that our adapted scheme outperforms both the original optimistic scheme and the existing iterated scheme [Smith et al. 2004] for most of benchmark codes. Also, we propose two techniques for further decreasing spills by considering the relationship between register classes assigned to each node in IG and measuring the benefit from the aggressive coalescing strategy.

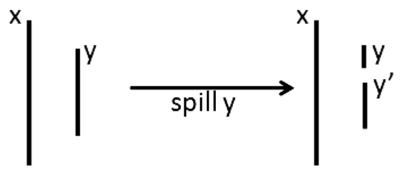


Fig. 28. A simple example of infinite spill.

In future work, we want to compare the performance of our optimistic coalescing and the register allocation algorithm with the optimal register allocation from the generic methods like integer linear programming.

APPENDIX

A. IMPLEMENTATION ISSUES

This appendix provides some issues about the implementation of the register allocation and coalescing algorithm. The detailed pseudocode about optimistic coalescing is described in Park et al. [2004]. Smith et al. [2004] deal with generalized graph coloring well. We want to explain one thing about the way of avoiding infinite spilling, which happens frequently and its debugging is a little bit complicated. In infinite spilling, a node is infinitely spilled with no squeezes in IG changed. For example, as shown in Figure 28, we assume that a variable **y** is spilled whose live range is included by the other live range of a variable **x**. It produces spill codes but **x** still has an interfering edge with **y** in IG, s the next round of register allocation, **y,** is marked as spills again, so on.

Most previous works briefly deals with this. They propose that the register allocator should be careful for not choosing the short live range generated from the spills in previous register allocation round. Chaitin [1982] and Bergner [1997] also note that spill code should not be generated in some cases such that a definition is 'close' to its use. For most of the cases including the example in Figure 28, they are good advices to avoid infinite spilling. We implemented Chaitin's spilling heuristic for minimizing spills. But when we implement our register allocation and optimistic coalescing, we found a case where infinite spills are still generated. It is because a node spilled in previous round of register allocation is coalesced again in next coalescing round. We can avoid this if we do not coalesce the nodes previously spilled in the aggressive coalesce stage described in Figure 5(b).

ACKNOWLEDGMENTS

This research is based on earlier work [Ahn et al. 2007] in which the work of Jooyeon Lee was pivotal. We thank him for his time and effort.

REFERENCES

AHN, M., LEE, J., JUNG, S., YOON, J. W., AND PAEK, Y. 2007. A code generation approach for Heterogeneous register architectures. In *Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*. IEEE, Los Alamitos, CA.

ARAUJO, G. AND MALIK, S. 1998. Code generation for fixed-point DSPs. *ACM Trans. Des. Automat. Electr. Syst. (TODAES) 3*, 2, 136–161.

BERGNER, P. E. 1997. Spill Code Minimization Techniques for Graph Coloring Register Allocators. PhD thesis, Minnesota University, Minneapolis, Minnesota.

BRIGGS, P. 1992. Register allocation via graph coloring. PhD thesis, Rice University, Houston, TX.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction.* ACM, New York, 98–105.

DAVEAU, J-M., THERY, T., LEPLEY, T., AND SANTANA, M. 2004. A retargetable register allocation framework for embedded processors. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, New York, 202–210.

FEUERHAHN, H. 1988. A data-flow driven resource allocation in a retargetable microcode compiler. In *Proceedings of the 21th Annual Workshop on Microprogramming and Microarchitecture*. IEEE, Los Alamitos, CA, 105–107.

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

KOES, D. AND GOLDSTEIN, S. C. 2005. A progressive register allocator for irregular architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, New York, 269–280.

KONG, T. AND WILKEN, K. D. 1998. Precise register allocation for irregular architectures. In *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, New York, 297–307.

LEE, C., POTKONJAK M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 330–335.

LEE, J. K., CHEN, S. Y., AND WU, C. J. 2006. Copy propagation optimizations for VLIW DSP processors with distributed register files. In *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC).* Springer, Berlin, Germany.

LIEM C., MAY T., AND PAULIN P. G. 1994. Register assignment through resource classification for ASIP microcode generation. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE, Los Alamitos, CA, 397–402.

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

PAULIN P. G., LIEM, C., MAY, T. C., AND SUTARWALA, S. 1995. DSP design tool requirements for embedded systems: a telecommunications industrial perspective. *J. VLSI Signal Process. Syst. 9*, 1-2, 23–47.

STALLMAN, R. M. 1994. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, MA.

SCHOLZ, B. AND ECKSTEIN, E. 2002. Register allocation for irregular architectures. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*. ACM, New York, 139–148.

SMITH, M. D., RAMSEY N., AND RAMSEY G. 2004. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 277–288.

ZIVOJNOVIC, V., VELARDE, J. M., AND SCHLAGER, C. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology.* IEEE, Los Alamitos, CA.

ZIVOJNOVIC, V., PEES, S., SCHLAGER, C., WILLEMS, M., SCHOENEN, R., AND MEYR, H. 1996. DSP processor/compiler co-design: a quantitative approach. In *Proceedings of the 9th International Symposium on System Synthesis*. IEEE, Los Alamitos, CA,108.

Received October 2007; accepted July 2008