SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2014, Vol. 57 092104:1–092104:18
doi: 10.1007/s11432-014-5131-7

Acyclic orientation graph coloring for software-managed memory allocation

WANG Li[1]*, XUE JingLing[1,2] & YANG XueJun[1]

[1]*School of Computer, National University of Defense Technology, Changsha* 410073*, China;*
[2]*School of Computer Science and Engineering, University of New South Wales, Sydney* 100049*, Australia*

Received July 27, 2013; accepted December 11, 2013; published online June 9, 2014

Abstract This paper presents a novel compiler algorithm, called acyclic orientation graph coloring (AOG coloring), for managing data objects in software-managed memory allocation. The key insight is that software-managed memory allocation could be solved as an interval coloring problem, or equivalently, an acyclic orientation problem. We generalize graph coloring register allocation to interval coloring memory allocation by maintaining an acyclic orientation to the currently colored subgraph. This is achieved with some well-crafted heuristics, including Aggressive Simplify that does not necessarily preserve colorability and Best-Fit Select that assigns intervals (i.e., colors) to nodes by possibly adjusting the colors already assigned to other nodes earlier. Our algorithm generalizes and subsumes as a special case the classical graph coloring register allocation algorithm without notably increased complexity: it deals with memory allocation while preserving the elegance and practicality of traditional graph coloring register allocation. We have implemented our algorithm and tested it on Appel's 27921 interference graphs for scalars (augmented with node weights). Our algorithm outperforms Memory Coloring, the best in the literature, for software-managed memory allocation, on 98.64% graphs, in which, the gaps are more than 20% on 68.31% graphs and worse only on 0.29% graphs. We also tested it on all the 73 DIMACS weighted benchmarks (weighted graphs), AOG Coloring outperforms Memory Coloring on all of the benchmarks, in which, the gaps are more than 20% on 83.56% graphs.

Keywords graph coloring, memory coloring, interval coloring, acyclic orientation, software-managed memory

Citation Wang L, Xue J L, Yang X J. Acyclic orientation graph coloring for software-managed memory allocation. Sci China Inf Sci, 2014, 57: 092104(18), doi: 10.1007/s11432-014-5131-7

1 Introduction

Modern high-performance computer architectures continue to embrace more and more on-chip processing elements. Architects must ensure that memory bandwidth and latency are optimized to exploit the full benefits of all computational resources available. Utilizing a cache hierarchy has been the traditional approach to alleviating the memory bottleneck. Despite this great success, some deficiencies with cache memories are well-known. First of all, their complex hardware logic incurs high overhead in power consumption and area. Second, their simple application-independent management strategy does not benefit from some data access characteristics in many applications. Finally, their uncertain access latencies make it difficult to guarantee real-time performance.

*Corresponding author (email: liwang@nudt.edu.cn)

In contrast, fast software-managed on-chip memory, often referred to as scratchpad memory (SPM for short) has advantages in area, cost, and access speed [1]. It is thus widely adopted in embedded systems, stream architectures, and GPUs. In the case of supercomputers, software-managed on-chip memory is also frequently used, especially in their accelerators. Examples include Merrimac [2], Cyclops64 [3], Grape-DR [4] and Roadrunner [5].

Unlike cache, SPM is managed by software. The software must schedule explicitly the data transfers between SPM and off-chip memory, and orchestrate the data allocation to optimize utilization of the scarce on-chip memory, and reduce memory fragmentation to accommodate frequently accessed data objects.

Many compiler approaches for SPM allocation have been proposed. These approaches can be roughly divided into two classes – those that resort to ILP [6] and those that rely on some heuristics [7–12]. The ILP-based approaches are theoretically optimal but too expensive to be practical for many applications. Among the heuristics-based approaches, memory coloring [8] seems to achieve the best performance for general-purpose applications [10]. However, memory coloring may introduce memory fragmentation, resulting in sub-optimal solutions as will be detailed in Section 4.

In this paper, we present a generalization of graph coloring register allocation for software-managed memory allocation. The key insight is that SPM allocation can be formulated and solved by interval coloring on the traditional interference graphs (IGs) except that an IG here is a weighted graph formed by the data objects to be placed in SPM. As graph coloring is a special case of interval coloring, we therefore generalize graph coloring register allocation to interval coloring memory allocation by constructing incrementally an acyclic orientation of a given IG.

In summary, this paper makes the following contributions:

1. We propose for the first time to optimize utilization of software-managed memory by acyclic orientation graph (AOG) coloring and present a smooth generation of traditional graph coloring while preserving its elegance and practicality. This is accomplished by using some well-chosen heuristics, including Aggressive Simplify that does not necessarily preserve colorability and Best-Fit Select that assigns intervals (i.e., colors) to nodes by possibly adjusting the colors already assigned to other nodes.

2. We show that our AOG algorithm (incorporating Best-Fit Select) yields better solutions for weighted IGs than two representative algorithms, i.e., First-Fit bin-packing and memory coloring.

The rest of this paper is organized as follows. For background information, Section 2 reviews the related work. Section 3 introduces existing graph coloring allocators for register and memory allocation and some basic results about interval coloring required to understand our approach. Section 4 motivates this work by an example. Section 5 describes our AOG coloring framework. Section 6 evaluates our approach. Section 7 concludes the paper.

2 Related work

There are a number of research efforts on SPM allocation. Existing approaches for SPM allocation can be roughly divided into two classes – those that resort to ILP [6] and those that rely on some well-crafted heuristics [7–10]. The ILP-based approaches are theoretically optimal but too expensive to be practical for many applications. Among the heuristics-based approaches, graph coloring [8] seems to achieve the best performances for general-purpose applications [10].

Graph coloring is a popular technique used in register allocation. Based on Chaitin's original formulation [13], a variety of graph coloring based register allocators have been developed [14–16]. In particular, George et al. [16] introduce a well-known iterative-coalescing algorithm. Smith et al. [17] present a generalized algorithm for irregular architectures with register aliases and non-disjoint register classes. Li et al. [8–11] apply it to assign arrays in embedded programs to scratchpad memory (SPM). Wang et al. [18–20] apply it further in stream register file allocation for stream processors.

Fabri [21] discovered the connection between interval coloring and compile-time memory allocation. However, interval coloring is NP-complete even when restricted to interval graphs (a class of so-called perfect graphs) with vertex weights in $\{1, 2\}$ [22]. Since then, the research of applying interval coloring

to compile-time memory allocation focuses on straight-line programs, i.e, programs without loops or conditional statements, in which case, their IGs are interval graphs. A number of approximation algorithms have been proposed [23–25], with the best one having a constant factor of $2 + \varepsilon$. However, the upper bound from mathematical analysis remains too conservative to be practically useful in computer science applications. Furthermore, straight-line programs (interval graphs) are too limited to be directly applied to real-world computer programs. In addition to compile-time memory allocation, Lefebvre and Feautrier [26] use interval coloring to minimize the number of data structures to rename in storage management for parallel programs. Li et al. [9] apply interval coloring to assign arrays in embedded programs to SPM.

Due to the one-to-one correspondence between interval colorings and acyclic orientations [27], enumerating acyclic orientations for a given graph is another way to solve the interval coloring problem. Ref. [28] presented the first algorithm to generate all the acyclic orientations of an arbitrary graph. Ref. [29] proposed another algorithm with lower complexity. Ref. [30] discovered that the number of acyclic orientations in a graph can be derived from its chromatic polynomial. Unfortunately, performing the acyclic orientation enumeration for an arbitrary graph is exponential [31].

3 Background

We review George and Appel's graph coloring register allocation algorithm in Subsection 3.1 and describe a recent generalization in Subsection 3.2. We review memory coloring in Subsection 3.3. In Subsection 3.4, we describe the key relevant results in interval coloring.

3.1 Graph coloring register allocator

Register allocation is one of the most important and well-studied compiler optimizations. Graph coloring is a popular and powerful technique for register allocation. George and Appel's iterated-coalescing algorithm is a textbook example [16]. Given K registers, it proceeds in the following phases:

• Build. An IG (interference graph) is constructed using the results of data flow analysis. Each node represents a variable and an edge connects two nodes if the variables represented by the nodes interfere and cannot be allocated to the same register.

• Simplify. This phase works based on Theorem 1 given below, which leads naturally to a stack-based algorithm for coloring: repeatedly remove (and push on a stack) nodes of degree less than K. Each such simplification will decrease the degrees of other nodes, leading to more opportunity for simplification. This is repeated until none of the remaining nodes can be simplified. If all nodes have been pushed on the stack, the algorithm skips to the Select phase.

• Potential Spill. If only nodes with degree greater than K are left, the algorithm marks a node as a potential spill node, removes it from the graph, and optimistically pushes it onto the stack in the hope that it might be able to be colored later. This process is repeated until there exist nodes in the graph with degree less than K, at which point, it returns to the Simplify phase.

• Select (assigns colors to nodes). Starting with the empty graph, the original IG is built up by repeatedly adding a node from the top of the stack. When a node is added to the graph, a valid color is chosen for it. If no valid colors are available, which happens only on potential spill nodes, the current node is marked as an actual spill, and the algorithm skips to the Actual Spill phase.

• Actual Spill. If any nodes are marked as actual spills, it generates spill code which loads and stores the variable represented by the node into new, short lived, temporary variables where the variable is used and defined. As new variables are introduced, it is necessary to rebuild the IG and repeat the above phases.

Theorem 1. Suppose a graph $\mathcal{G}$ contains a node m with fewer than K neighbors, where K is the number of registers on the machine. Then $\mathcal{G}$ is K-colorable if and only if $\mathcal{G}$ with m (and its incident edges) removed is K-colorable.

There is a phase, Coalesce. We do not list it here, nor do we show how it fits into our framework, because it aims to eliminate unnecessary copy operations, and our generalization neither makes any revision to its process, nor have any influence on how it fits into our new framework and how it works.

3.2 Generalized graph coloring

The traditional algorithm described in Subsection 3.1 is based on two assumptions: registers are expected to be interchangeable and independent. Registers are interchangeable if they are equally suitable in any program context. Registers are independent if writing to one cannot change the value of another. However, these two assumptions are invalid for most commercial instruction sets. Smith et. al present [17] a generalization of the traditional trivial colorability criterion, which is capable of measuring the impact on colorability of a node imposed by its neighbors. For a node n, it is trivially colorable provided $\text{squeeze}_n^* < |\text{class}_n|$, where squeeze_n^* is the maximum number of names from class_n that could be denied to n because of an assignment of registers to n's current neighbors, and $|\text{class}_n|$ is the number of registers in the class of n. Thus, squeeze_n^* is conservatively approximated as follows:

$$\text{squeeze}_n^* = \sum_{C \in \mathbb{C}} \text{degree}_n(C) * \text{worst}^1(\text{class}_n, C), \tag{1}$$

where $\mathbb{C}$ is the set of all registers classes that n's neighbors belong to, $\text{degree}_n(C)$ is the number of node n's neighbors in class C, and $\text{worst}^1(N, C)$ is the worst-case number of registers of class_n that can alias with 1 element of class C. Note that $\text{worst}^1(N, C)$ represents a constant that depends only on the properties of the target architecture. This generalization permits simultaneous allocation of multiple register classes, even when registers alias.

3.3 Memory coloring

The efficiency of generalized graph coloring register allocation has led to its adaptation in software-managed memory allocation, called Memory Coloring [10]. By partitioning SPM to create pseudo register classes to hold the data objects in a program, their automatic placement in SPM can be achieved by graph coloring. This approach seems to achieve better solutions for managing arrays in general-purpose applications than other heuristics [10].

There are two major steps: array normalization and memory partition. Array normalization clusters the arrays in a program into array classes so that the arrays in the same class have the same aligned (or normalised) size. A tunable parameter, *ALIGN_UNIT*, is used to avoid introducing a large number of array classes containing arrays with similar sizes, resulting in an unnecessarily large register file. In memory partition, the space of a given SPM is partitioned multiple times to create the (pseudo) register classes, one register class per array class. By partitioning the SPM multiple times, aliases are introduced between registers. Two registers (in different classes) are aliases if their SPM spaces overlap and independent otherwise. Two registers (in the same register class) are interchangeable if they have the same size but disjoint SPM spaces. After both steps are performed, the generalized graph coloring algorithm reviewed in Subsection 3.2 is applied to place the arrays in SPM.

3.4 Interval coloring

3.4.1 *Notations and terminologies*

We denote an unweighted graph by $\mathcal{G} = (V, E)$, where V is the node set and E is the edge set. The neighborhood $N(x)$ of a node x in $\mathcal{G}$ is the set of all vertices of $\mathcal{G}$ that are adjacent to x. A weighted graph $(\mathcal{G}; w)$ associates each node $x \in V$ with a strictly positive integer weight w_x. A K-interval coloring of $(\mathcal{G}; w)$ is a function I that assigns an interval $I(x) \subseteq \{1, \ldots, k\}$ of w_x consecutive integers (called colors), to each node $x \in V$, such that for each edge $(x, y) \in E$, $I(x) \wedge I(y) = \emptyset$. The interval coloring problem (ICP) is how to determine the smallest integer K, called chromatic number of $(\mathcal{G}; w)$ and denoted by $\chi_{\text{int}}(\mathcal{G}; w)$, such that there exists a K-interval coloring of $(\mathcal{G}; w)$. For a fixed integer K, the K-interval graph coloring problem (K-ICP) is how to determine a K-interval coloring of $(\mathcal{G}; w)$. The ICP and k-ICP

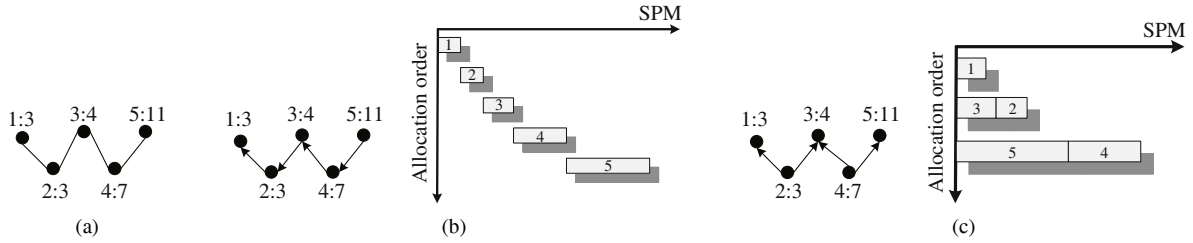

Figure 1 Equivalence between acyclic orientations and interval colorings ($n : w(n)$ denotes node n has weight $w(n)$). (a) $(\mathcal{G}; w)$; (b) $\chi_\alpha(\mathcal{G}; w) = 28$; (c) $\chi_\beta(\mathcal{G}; w) = 18$.

generalize classical node coloring problems where a single color is assigned to each node (i.e., $w_x = 1$ for all nodes $x \in V$).

To avoid cluttering, $(\mathcal{G}; w)$ is abbreviated to $\mathcal{G}$ and $\chi_{\text{int}}(\mathcal{G}; w)$ to $\chi(\mathcal{G})$. In addition, K-interval coloring is shortened to K-coloring whenever no confusion arises.

3.4.2 *Interval coloring vs. acyclic orientation*

Let $\mathcal{G} = (V, E)$ be an undirected graph. An *orientation* of $\mathcal{G}$ is a function α that assigns every edge a direction such that $\alpha(x, y) \in \{(x, y), (y, x)\}$ for all $(x, y) \in E$. Let $\mathcal{G}_\alpha$ be the digraph obtained by replacing each edge $(x, y) \in E$ with the arc $\alpha(x, y)$. An orientation α is said to be *acyclic* if $\mathcal{G}_\alpha$ contains no directed cycles.

Interval coloring could be understood from a different perspective. Every interval coloring α of $\mathcal{G}$ induces an acyclic orientation α' such that $(x, y) \in \alpha'$ if and only if α_x is to the right of α_y for all $(x, y) \in E$. Conversely, an acyclic orientation α of $\mathcal{G}$ induces an interval coloring α'. To see this, for a sink node x (without successors), let $\alpha'_x = [0, w(x) - 1]$. Proceeding inductively, for a node y with all its successors already colored, let $\alpha'_y = [t, t + w(y) - 1]$, where t is the largest endpoint of their intervals.

Let us now define a height function h on V, $h(v) = t$, $\forall\ v \in V$, where t is the right endpoint of its interval. Thus, $h(v)$ represents the interval allocated to v. Given an acyclic orientation, the corresponding interval coloring, i.e., the heights of the vertices could be computed in linear time by a depth-first search.

The problem of finding optimal colorings is NP-complete. In an optimal coloring, the chromatic number $\chi(\mathcal{G}; w)$ is related to the notion of *heaviest path* in an acyclic orientation of $\mathcal{G}$ as follows:

$$\chi(\mathcal{G}; w) = \min_{\alpha \in \mathcal{A}(\mathcal{G})} (\max_{\mu \in \mathcal{P}(\alpha)} \text{w}(\mu)), \tag{2}$$

where $\mathcal{A}(\mathcal{G})$ is the set of all acyclic orientations of $\mathcal{G}$ and $\mathcal{P}(\alpha)$ the set of all directed paths in an orientation $\alpha \in \mathcal{A}(\mathcal{G})$. In other words, the orientation whose heaviest path is the smallest induces an optimal coloring. This heaviest-path-based formulation is exploited in the development of our interval coloring memory allocator.

Figure 1 illustrates the equivalence between finding an interval coloring and finding an acyclic orientation for a weighted graph. For the example graph shown in Figure 1(a), two different acyclic orientations, α and β, together with their corresponding SPM allocation results, are given in Figure 1 (b) and (c), respectively. In Figure 1(b), the heaviest path is $5 \to 4 \to 3 \to 2 \to 1$ with a (total) weight of $\chi_\alpha(\mathcal{G}; w) = 28$. In Figure 1(c), the heaviest path is $4 \to 5$ with a weight of $\chi_\beta(\mathcal{G}; w) = 18$. The gap between the two solutions is 10 but can be larger in general. So there is a need to look for an optimal solution efficiently in practice.

3.4.3 *Interval coloring vs. strict partial order*

A strict partial order $\prec$ is a binary relation on a set $\mathcal{P}$ that is irreflexive and transitive, and therefore, asymmetric.

A K-coloring α of an unweighted graph $\mathcal{G} = (V, E)$ defines a strict partial order $\prec$ on V as follows. Let us number the colors as $\{1, 2, \ldots, k\}$. Then α induces a function $f : V \to \{1, 2, \ldots, k\}$, that is, $\forall v \in V$, $f(v)$ is the color assigned to v. Define a binary relation $\prec_f$ on V such that $\forall u, v \in V$, $u \prec_f v$ if and only

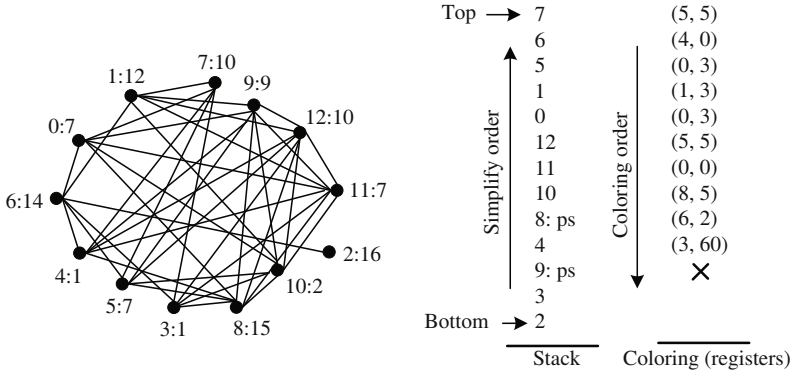

Figure 2 Example IG and its memory coloring ($K = 61$). (a) IG (Graph #47 [1]); (b) Node stack and coloring.

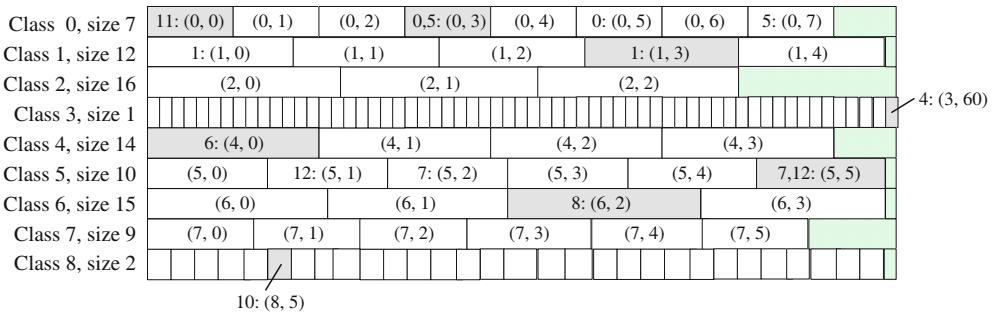

Figure 3 The register file for an SPM of size $K = 61$ created by memory coloring [10] for the IG in Figure 2 ($ALIGN\_UNIT = 1$). The allocation result is highlighted by shaded registers.

if $f(u) < f(v)$. Obviously, $\prec_f$ is a strict partial order. Intuitively, a coloring defines a strict partial order on the adjacent vertices in $\mathcal{G}$.

A strict partial order $\prec_f$ on the node set V of a graph $\mathcal{G} = (V, E)$ induces an acyclic orientation of $\mathcal{G}$ as follows. $\forall (u, v) \in E$, if $u \prec_f v$, direct the edge from v to u, otherwise from u to v.

All these discussions apply to interval coloring if we define the order between two intervals I_1 and I_2 to be, $I_1 < I_2$ if and only if the end address of I_1 is smaller than the start address of I_2.

4 Motivation

Memory coloring, while performing better than other heuristics in SPM allocation [10], has some deficiencies. Let us use an example to reveal these and motivate the development of our approach.

Figure 2(a) depicts Graph #47 from Appel's IG collection for scalars[1)]. For initially scalar, the IG is turned into a weighted graph by assigning to the nodes the weights chosen randomly according to a discrete uniform distribution over [1,16]. In the figure, $n : w(n)$ signifies that node n has a weight of $w(n)$. Let the SPM size be $K = 61$. According to [10], the pseudo register file obtained is depicted in Figure 3 when $ALIGN\_UNIT = 1$. There are nine register classes created for nine different data sizes. A register is identified by a pair (c, r), where c is its class number and r its position in class c. The solution obtained is shown by the shaded registers in Figure 3 and explained briefly below.

Given the register architecture in Figure 3, $\text{worst}^1(M, C)$ introduced in Subsection 3.2 can be calculated as given in Table 1. For example, $\text{worst}^1(0, 1) = 3$, because one register in class 1 may alias with at most 3 registers in class 0. To see this, note that register $(1, 1)$ aliases with registers $(0, 1)$, $(0, 2)$ and $(0, 3)$.

During the Simplify phase, memory coloring tries to simplify nodes in a certain order, say, in increasing order of their node numbers. Let class_n be the register class for node n as before. For node 0, its

1) Andrew W. Appel. Sample graph coloring problems. 1996. http://www.cs.princeton.edu/ appel/graphdata.

Figure 4 The iterated AOG coloring framework.

Table 1 Register class aliasing table for $\text{worst}^1(M, C)$

Class	0	1	2	3	4	5	6	7	8
0	1	3	3	1	2	3	3	3	2
1	2	1	2	1	2	2	2	2	1
2	2	2	1	1	2	2	2	2	1
3	7	12	16	1	14	10	15	9	2
4	1	2	2	1	1	2	2	2	1
5	2	2	3	1	3	1	2	2	1
6	2	2	2	1	2	2	1	2	2
7	2	2	3	1	3	2	3	1	2
8	4	6	8	1	7	5	8	5	1

neighborhood is $N(0) = \{6, 7, 8, 9, 10, 11\}$. According to (1), $\text{squeeze}_0^* = \sum_{x \in N(0)} \text{worst}^1(\text{class}_0, \text{class}_x) = 14$. However, the number of available registers in class 0 is only $|\text{class}_0| = 8$. Since $\text{squeeze}_0^* > |\text{class}_0|$, node 0 cannot be simplified. The remaining nodes are processed similarly. If no node can be simplified, a node with the highest degree and the smallest weight is chosen to be a potential spill node. The final node stack is shown in Figure 2(b), where 8 and 9 are potential spill nodes. Next, the nodes are evicted from the stack one by one, and a register with the smallest number of aliases is assigned to it, resulting in the coloring/allocation shown in Figure 2(b) and also further illustrated in Figure 3. For node 9, no valid register is available for it. In fact, the minimum SPM size needed under memory coloring is 62 (1 unit larger than the SPM size given).

Some deficiencies of memory coloring are as follows.

1. Aliasing between registers in different classes may cause inter-register memory fragmentation. In Figure 3, register $(8, 7)$ of size 2 with a start address of 14 aliases with registers $(6, 0)$ and $(6, 1)$ of size 15. Assigning (8,7) to an object will consume a space of size 30 from the perspective of these two neighbors.

2. Array normalization introduces intra-register memory fragmentation. The larger *ALIGN_UNIT* is, the worse the situation is. However, small *ALIGN_UNIT* values may create too many register classes, causing inter-register memory fragmentation.

3. Memory partition may introduce unused gaps at the "right end" of memory for some register classes as shown in Figure 3 when the memory size is not a multiple of their corresponding object sizes.

However, one major advantage with memory coloring is that it has successfully casted a continuous optimization problem in terms of interval coloring into a discrete problem, by limiting the placement of data objects at certain aligned addresses. As a result, the solution space has been greatly reduced. By virtue of the power of graph coloring algorithms, memory coloring has achieved so far the best performance results over other heuristics [10].

Inspired by the relationship between interval colorings and acyclic orientations, we generalize graph coloring register allocation to SPM allocation in a natural way. Our approach neither create register classes, nor perform array normalization and memory partition, thus it can fully avoids the deficiencies introduced by memory coloring, and it preserves the practicality and elegance of traditional graph coloring, and subsumes as a special case the classic graph coloring approach.

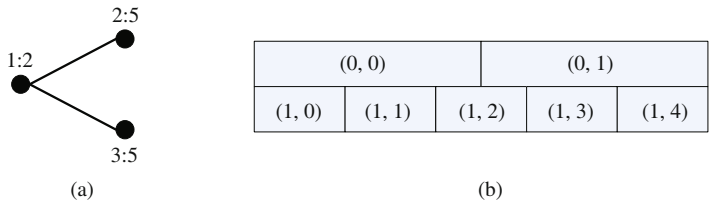

Figure 5 An example demonstrating the conservatism of the simplification criterion (1) in generalized graph coloring [17]. (a) IG; (b) the register file.

5 The AGO coloring framework

Our AOG coloring framework is given in Figure 4. Our generalization from graph coloring register allocation to memory allocation is reflected in the phases, Aggressive Simplify, Potential Spill and Best-Fit Select. We do not discuss coalescing since our generalization neither makes any revision to its process, nor has any influence on how it fits into our new framework and works.

5.1 Aggressive simplify

This Simplify phase is first proposed by Chaitin [13] in the first practical graph coloring register allocator, with the aim to repeatedly simplify a given IG while preserving its colorability. In Chaitin's design, if a trivially colorable node cannot be found, a node is selected to spill. Briggs et al. later [15] pointed out that Chaitin's allocator gives up coloring too early. In contrast, when the current graph has no nodes that can be simplified, they select a node, mark it as a potential spill node, and then push it on the stack. Their key idea is to postpone the spill decision to the Select phase. A node is spilled only when no valid color can be assigned to it in Select. In other words, all nodes get a chance to queue on the stack to be colored.

The Potential Spill phase thus introduced reduces the pressure imposed on the Simplify phase for preserving colorability; it works by contributing to the process of constructing a coloring order for the nodes in the IG, which is actually the reverse order of node simplification. In general, it is difficult to judge the quality of a coloring order for an IG. In fact, different coloring orders have no major impact on the coloring quality, as validated in our experiments. The only exception is the programs in SSA form, in which case the IGs are chordal graphs, a special class of so-called perfect graphs [27]. For chordal graphs, if the coloring order obeys the Perfect Elimination Order (PEO) [32], using a First-Fit-like greedy coloring heuristic will always yield an optimal solution.

The simplification criterion (1) developed in generalized graph coloring (Subsection 3.2), and consequently, in memory coloring, is more conservative than its counterpart used in traditional graph coloring (Theorem 1), and sometime too conservative to make many colorable nodes potential spilled. Consider the IG and the register file shown in Figure 5, where $\text{worst}^1(0,1) = 2$ and $\text{worst}^1(1,0) = 3$. Based on (1), none of them can be simplified. However, the IG is obviously colorable.

As described above, any node, potential spilled or not, will get a chance to be assigned a color in the Select phase, motivating us to relax the restriction that the simplification criterion must preserve colorability and propose an aggressive simplify criterion.

Let k be the size of the SPM. For a weighted graph $(\mathcal{G}; w)$, where $\mathcal{G} = (V, E)$, let $w(N(x)) = \sum_{y \in N(x)} w(y)$, i.e., the sum of weights of the neighbors of node x. Our algorithm repeatedly removes a node x satisfying $w(x) \leqslant k - w(N(x))$ in the current subgraph of $\mathcal{G}$ and pushes x on a stack. This is repeated until none of the remaining nodes satisfies the condition. If all nodes have been simplified, the algorithm skips to the Best-Fit Select phase.

In the context of register allocation, k denotes the number of available registers. So $w(x) = 1$ for all $x \in V$. Thus, $w(x) \leqslant k - w(N(x))$ simplifies to $w(N(x)) \leqslant k - 1$, $w(N(x)) = \sum_{y \in N(x)} w(y)$, $\forall y \in N(x), w(y) = 1$, thus, $w(N(x)) = |N(x)|$, i.e., $|N(x)| \leqslant k - 1$, which is exactly the simplification criterion used in the standard graph coloring register allocation framework (Theorem 1). Thus, our criterion represents a seamless generalization.

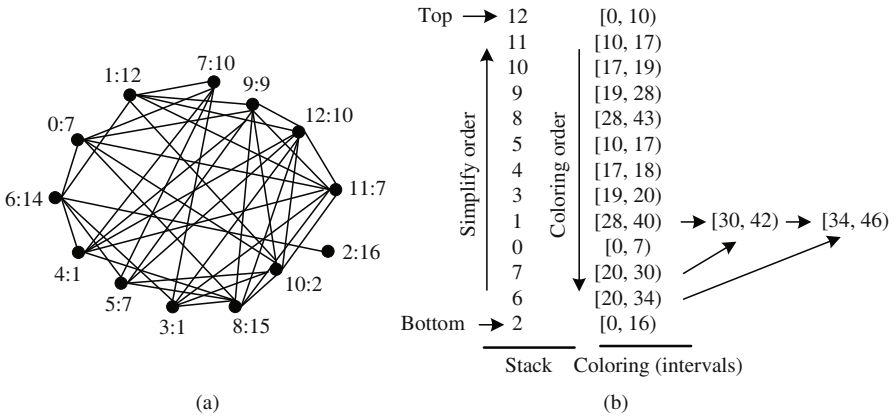

Figure 6 Same IG in Figure 2 and its AOG coloring ($K = 46$). (a) IG (Graph #47 [1]); (b) Node stack and coloring.

However, our generalized simplification criterion is not colorability-preserving. That is, if a node x satisfies $w(x) \leqslant k - w(N(x))$ and $\mathcal{G}[V - \{x\}]$ is k-interval colorable, then $\mathcal{G}[V]$ is not necessarily k-interval colorable. In this case, the neighbors of x may sum up to at most $w(N(x))$, implying that the sum of all the gaps formed by the neighbors of x is not smaller than $w(x)$. However, every gap may be smaller than $w(x)$! Recall that interval coloring requires a continuous memory space for each node. For example, take a look at the allocation in Figure 1(b), suppose the SPM size $k = 7$, and the coloring order of the nodes is $1, 2, 3, 4, 5$, then before coloring 3, there are two gaps formed by its neighbor 2, $[0, 3)$ and $[6, 7)$, the sum of the two gaps is 4, not smaller than its weight $w(3) = 4$, however, each gap is not large enough to fit it in. Nevertheless, we optimistically assume that in most cases, if a node x satisfies $w(x) \leqslant k - w(N(x))$, then x is colorable, namely, there will be a gap larger enough to accommodate x when x is colored. That is why we have named phase Aggressive Simplify.

Let us perform Aggressive Simplify on the IG shown in Figure 6(a), the same IG shown in Figure 2(a), again in increasing values of their node numbers. Suppose the SPM size is $k = 46$. We compute progressively that $w(0) = 7 > k - w(N(0)) = 46 - 57$, $w(1) = 12 > k - w(N(1)) = 46 - 52$ and $w(2) = 16 < k - w(N(2)) = 46 - 14$. So node 2 is selected to be simplified. Similarly, the remaining nodes are simplified in the following order $6, 7, 0, 1, 3, 4, 5, 8, 9, 10, 11, 12$. No nodes are potentially spilled.

5.2 Potential spill

If there are no nodes available for simplification, we select a node x such that $w(N(x))$, the weighted node degree of x, is the largest and push it on the stack. In the case of a tie, the node with the smallest node weight itself is selected to minimize the spill cost. This is a smooth generalization from classical register allocation, in which case, a node with the highest node degree, denoted by $w(N(x))$, which simplifies to $|N(x)|$, is potentially spilled first.

Other factors such as the profiled access frequencies of nodes may also be taken into account straightforwardly.

5.3 BEST-Fit select

Like the traditional Select phase, Best-Fit Select repeatedly pops a node x from the stack and assigns a valid color range (or an interval) to it. The difference lies in how colors are selected. The traditional process randomly picks an available color range to x, which we call Random-Fit Select. Another frequently used heuristic is First-Fit Select, which always tries to place the current coloring candidate at the left-most possible address in SPM.

In the context of interval coloring, Random-Fit is obviously less effective than First-Fit, since Random-Fit may accidentally drop a node in the middle of a big gap, introducing two memory fragments at each end. In contrast to these two heuristics, Best-Fit looks for a gap I formed by x's colored neighbors for x to be placed into such that the total increase in the chromatic number of the current colored subgraph,

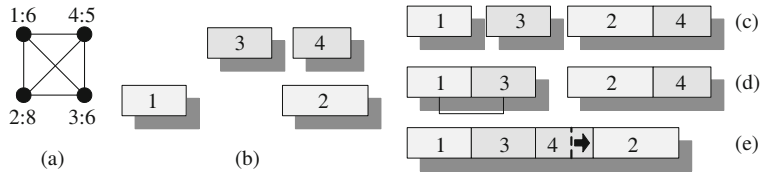

Figure 7 An example comparing the three coloring heuristics. (a) IG; (b) before allocating $\{3, 4\}$; (c) Random-Fit; (d) First-Fit; (e) Best-Fit.

including x, is minimized. In particular, suppose the address of the right end of the x's rightmost neighbor is p, such that $p < k$, where k is the SPM size. Then the interval $[p, k)$ is also regarded as a candidate gap, called the rightmost gap. All other gaps are called normal gaps.

Intuitively, Best-Fit tries to minimize the chromatic number of the current colored subgraph by minimizing its increase each time a node is colored. If there exists a normal gap to accommodate x, Best-Fit works exactly as First-Fit, by keeping the current chromatic number unchanged. However, both differ when no such normal gaps are available. In this case, First-Fit always places x in the rightmost gap. In contrast, Best-Fit compares all the small normal gaps (smaller than $w(x)$) as well as the rightmost gap and picks one gap such that placing x in the gap causes the least increase in chromatic number. In other words, First-Fit and Random-Fit never insert x into a gap smaller than its size but Best-Fit may do so by expanding the gap rightward to make room for x.

Let us look at the three coloring heuristics in more detail. First-Fit aims to put x into a gap I subject to two conditions. One is $w(I) \geqslant w(x)$ and the other is that the start address of I must be minimized (greedy fit). First-Fit never expands a gap. One main drawback is that placing a node inside a big gap may render the remaining free space unusable. In contrast, Best-Fit tries to utilize all gaps to minimize the total increase in chromatic number. Random-Fit shares the same problems as First-Fit and has also a problem of its own. Random-Fit may randomly place a node in the middle of a big gap, introducing two unusable memory fragments.

To reveal the deficiency of First-Fit more intuitively, let us assume there are a series of data objects, numbered $1, 2, 3, \ldots, n$, with their lengths being defined below:

$$\mathcal{L}_m = \begin{cases} a, & m = 1, \\ b, & m = 2, \\ \sum_{i=1}^{m-2} \mathcal{L}_i + 1, & 3 \leqslant m \leqslant n. \end{cases} \tag{3}$$

Suppose further that $a = 3$ and $b = 3$ and the IG when $n = 5$ is shown in Figure 1(a). Placing them in increasing order of their node numbers, First-Fit gives a very poor solution in Figure 1(b).

Let us consider another example in Figure 7 to illustrate how the three coloring heuristics work. For the IG shown in Figure 7(a), there are four nodes in $\{1, 2, 3, 4\}$ to be colored, say, by increasing order of their node numbers. As shown in Figure 7(b), nodes 1 and 2 have already been colored, with node 1 placed at the start address 0 and node 2 at 15. This leaves a gap of size 9 in between. When coloring node 3, all the three heuristics will put it into this gap. However, Random-Fit may randomly place it in the middle of the gap, resulting in two fragments as highlighted, while both First-Fit and Best-Fit align node 3 with the left end of the gap. When coloring node 4, the only gap(s) formed around nodes 2 and 3 are too small to fit node 4 in in each case. First-Fit puts node 4 to the right of node 2. Likewise for Random-Fit. In contrast, Best-Fit will insert 4 into the gap between nodes 2 and 3 by expanding the gap to the right, resulting in the best solution for this example.

Our algorithm for performing Best-Fit Select is given in Algorithm 1. In lines 4 and 5, we compute the normal gaps and the rightmost gap formed by the neighbors of node x to be colored. In line 7, the normal gaps are checked. If a gap I is large enough to accommodate x (line 8), x is placed at the start address of the gap (line 9). The height of x is updated (line 10). Finally, *orientation_assignment* is called to assign the orientations to the edges adjacent to x. This is exactly what First-Fit does under this situation. If every normal gap is too small to fit x squarely in, then all these gaps are checked again,

Algorithm 1 BEST-Fit Select

```
1: procedure BEST_Fit_Select(G, x)
2: Input: A colored subgraph G = (V, E) of G and a node x
3: Output: The chromatic number χ_c of G after x is colored
   // Let s(v) be the start address of a node v
   // Let h(v) be the height of a node v
4: Let V_nc ⊆ V be the set of the colored neighbors of x
5: Let S_gap be the set of normal and rightmost gaps formed by V_nc
6: Let s(I) be the start address of gap I ∈ S_gap
7: for each normal gap I ∈ S_gap in increasing order of start addresses do
8:    if |I| ⩾ w(x) then
9:       s(x) = s(I)
10:      h(x) = s(x) + w(x)
11:      orientation_assignment(G, x)
12:      return χ_c
13:   end if
14: end for
15: Let χ_c be the chromatic number of G, i.e., χ_c = max_{v∈V} h(v)
16: χ_best = +∞
17: for each gap I ∈ S_gap in increasing order of start addresses do
18:   s(x) = s(I)
19:   orientation_assignment(G, x)
20:   χ_cur = height_update(G, x)
21:   if χ_cur == χ_c then
22:      return χ_c
23:   else if χ_cur < χ_best then
24:      χ_best = χ_cur
25:      I_best = I
26:   end if
27: end for
28: s(x) = s(I_best)
29: orientation_assignment(G, x)
30: height_update(G, x)
31: return χ_best
32: end procedure
```

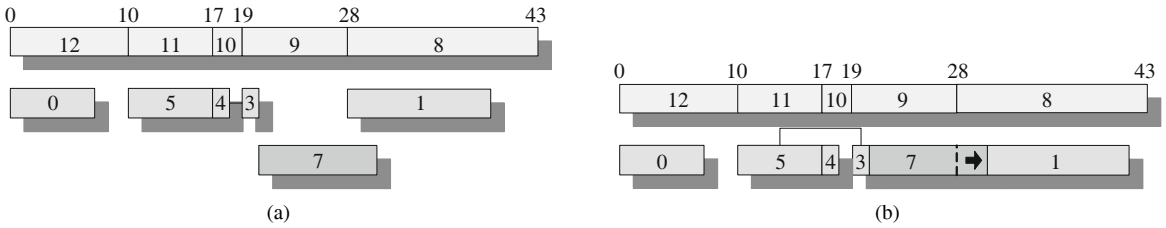

Figure 8 Coloring of node 7 (3rd from the last in stack) in Figure 6 ($K = 46$). All nodes except 6 and 2 have been colored as shown. (a) Before; (b) after.

together with the rightmost gap (line 17). For each gap I, we tentatively place x inside (line 18) and then invoke *orientation_assignment* and *height_update* in order to calculate the new chromatic number of $\mathcal{G}$ after x is colored (lines 19 and 20). If the chromatic number remains unchanged, we are done (line 22). Otherwise, the leftmost gap that gives rise to the smallest increase in chromatic number is used (lines 23–31).

For the node stack shown in Figure 6(a), Figure 8 illustrates Best-Fit Select in action in coloring node 7. All those above node 7 in the node stack have been colored as shown. The chromatic number of the current colored subgraph is $\chi_c = 43$. In line 4, we find that $V_{\rm nc}(7) = \{0, 1, 3, 4, 5\}$. The gaps $S_{\rm gap}$ formed by $V_{\rm nc}(7)$ are $\{[7, 10), [18, 19), [20, 28), [40, \infty)\}$, where $[40, \infty)$ is the rightmost gap and the others are normal gaps. Obviously, each normal gap is smaller than node 7. In line 17, each normal gap is checked again, together with the rightmost gap. Let us write χ_I to represent the chromatic number of the new subgraph after node 7 has been inserted into gap I. Then $\chi_{[7,10)} = 50$, $\chi_{[18,19)} = 44$ and

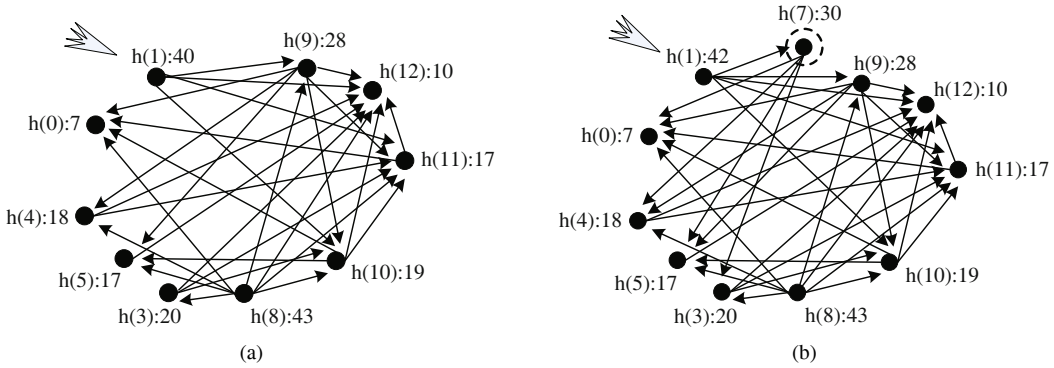

Figure 9 Acyclic orientations before and after node 7 is colored for the current colored subgraph shown in Figure 8. (a) Before; (b) after.

$\chi_{[20,28)} = 43$. The gap $[20, 28)$ is selected since placing node 7 inside does not increase the chromatic number. Being smaller than $w(7)$, the selected gap is expanded to accommodate node 7. Thus, the height of node 1 has been updated, as illustrated in Figure 8.

5.4 Orientation assignment

Best-Fit Select assigns an interval to the current coloring candidate x, thereby imposing a strict partial order between x and its colored neighbors. However, due to the difference between Best-Fit Select and the traditional Select phase, the intervals already assigned to some nodes may change, i.e., they may be pushed rightward to make room for x. However, the strict partial order itself does not change.

Therefore, we have introduced this new phase to record the strict partial order to facilitate the computation of the gaps required in line 5 of Best-Fit Select during the ensuing iteration. Due to the equivalence between strict partial order and acyclic orientation as discussed in Subsection 3.4, this phase maintains an acyclic orientation of the current colored subgraph incrementally.

Algorithm 2 Orientation Assignment

```
1: procedure orientation_assignment(G, x)
2: Input: A colored subgraph G = (V, E) of G and a node x
3: Let s(v) be the start address of a node v
4: Let V_nc ⊆ V be the set of colored neighbors of x
5: for each y ∈ V_nc do
6:     if s(x) < s(y) then
7:         Let the orientation to edge (x, y) be y → x
8:     else
9:         Let the orientation to edge (x, y) be x → y
10:    end if
11: end for
12: end procedure
```

Given a node x to be colored, this phase works as follows. For each $y \in V_{\text{nc}}$, if x is placed to the left side of y, i.e., $I_x < I_y$ or $x \prec y$, orient the edge (x, y) as $y \to x$, otherwise $x \to y$.

Continuing from Figure 8, Figure 9 shows the acyclic orientations before and after node 7 is colored.

In essence, once this phase is entered, Best-Fit Select has determined a strict partial order between the current node x being colored and its colored neighbors. This order is represented and maintained by an acyclic orientation in the current colored subgraph $\mathcal{G}$. For each neighbor $y \in N(x)$, if the start address of x is smaller than y, i.e., x is placed to the left of y, we let the orientation of (x, y) be $y \to x$; Otherwise, $x \to y$.

5.5 Height update

Best-Fit Select assigns an interval I_x to the current coloring candidate x, represented by a height value

of x, $h(x) = S(I_x) + w(x)$, where $s(I_x)$ is the start address of I_x. If no node needs to be moved to make room for x in the Best-Fit Select phase, this phase is not needed, since the heights of all colored nodes remain unchanged. Otherwise, the heights of some nodes, or more precisely, some of x's colored predecessors (immediate or non-immediate) need be recalculated. Based on the acyclic orientation constructed in *orientation_assignment*, this can be done by calling *height_update* in $O(n)$ time as discussed in Subsection 3.4.2, where n is the number of colored nodes. The height of each node represents the interval assigned to it. So the height information will be used to compute the gapes in line 5 of Best-Fit Select during the ensuing iteration.

In Figure 9, the heights of the nodes before node 7 is colored are shown in Figure 9(a). After node 7 has been colored, the height of node 1 has increased from 40 to 42 as shown in Figure 9(b), since node 1 has been pushed rightward to make room for node 7.

Algorithm 3 Height Update

1: **procedure** *height_update*$(\mathcal{G}, x)$
2: **Input:** A colored subgraph $\mathcal{G} = (V, E)$ of $\mathcal{G}$ and a node x
3: **Output:** The chromatic number χ of $\mathcal{G}$
// Let $h(v)$ be the height of a node v
4: Recalculate the heights $h(y)$ for all (immediate or non-immediate) predecessors y of x based the acyclic orientation maintained for $\mathcal{G}$
5: **return** $\max_{v \in V} h(v)$

In register allocation, whenever a gap exists, its size can never be smaller than the size of the node being colored, which is always 1. Thus, other nodes will never have to be pushed rightward. Therefore, neither the *orientation_assignment* nor the *height_update* phases are needed, reducing our interval coloring memory allocation to the traditional graph coloring register allocation.

5.6 Actual spill

If the maximum height of a node is larger than the SPM in size, then the current node candidate must be spilled. The spill code is inserted in the normal manner, and the above process is repeated.

5.7 Complexity

The phases Aggressive Simplify and Potential Spill have the same complexity as their counterparts in traditional algorithm. Both the phases Orientation Assignment (Algorithm 2) and *height_update* (Algorithm 3) could be done in $O(n)$ time. For the phase Best-Fit Select as shown in Algorithm 1, the most complex part is from the loop in line 17, at the worst case, there are n gaps. It takes at most $O(n)$ time to calculate the chromatic number for each gap; thus it has a complexity of $O(n^2)$. However, in practice, the complexity is much lower since it will return directly when there is a big gap, or a gap such that the chromatic number keeping unchanged. The phase Actual Spill has the same complexity with the phase in traditional algorithm.

6 Experiments

We implemented the AOG Coloring algorithm, and tested it on the 27,921 interference graphs made publicly available by George and Appel[1)], with the interferences between precolored registers and temporaries not considered. While these graphs are originally unweighted, we assign random weights to nodes to turn them weighted. The weights are in the range $[1, 16]$ in discrete uniform distribution. As a comparison, we also implemented the memory coloring algorithm [10]. Furthermore, we tested the two approaches on the 73 DIMACS weighted benchmarks[2)], which are from the real-life applications or graph theory researches. We performed the experiments on a 2.66GHz Intel (R) Core (TM) 2 Quad CPU Q9400 with 2GB Memory. The longest time taken is 137 seconds for the #16599 IG from the George

2) Graph coloring and its generalizations. http://mat.gsia.cmu.edu/ color03/, 2003.

Table 2 Colorings found by AOG Coloring compared to those found by Memory Coloring with different $ALIGN\_UNIT$ for 27921 IGs

$ALIGN\_UNIT$	Better	Equal	Worse
1	27542 (98.64%)	297 (1.06%)	82 (0.29%)
2	27706 (99.23%)	168 (0.60%)	47 (0.17%)

Table 3 The gaps between AOG Coloring and Memory Coloring with different $ALIGN\_UNIT$ for 27921 IGs

$ALIGN\_UNIT$	$\leqslant -10\%$	$< 0\%$	0%	$> 0\%$	$\geqslant 10\%$	$\geqslant 20\%$	$\geqslant 30\%$	$\geqslant 40\%$	$\geqslant 50\%$	$\geqslant 60\%$	$\geqslant 70\%$	$\geqslant 80\%$	$\geqslant 90\%$	$\geqslant 100\%$
1	3	79	297	2211	6229	8095	5836	3210	1382	467	96	13	2	1
2	2	45	168	2116	6275	8861	6050	3044	1054	256	37	11	0	2

Table 4 Optimality of AOG Coloring and Memory Coloring for 1472 IGs

Total	AO	MC
1472	1295 (87.98%)	80 (5.43%)

and Appel's 27921 IGs with 7420 nodes and 57373 edges. For the graphs with node count from 100 to 1000, the time taken is from 0.001 to 1 second, For example, for the #22940 IG with 502 nodes and 6711 edges, the time taken is 0.09 second. For most of the other IGs (96% of the IGs), the node count is less than 100, and the time is less than 0.001 second each.

We compare two solutions α and β as follows. The quality of α found by one approach (heuristics) is measured as a gap with respect to β found by other approach (heuristics) defined as follows:

$$\text{gap}(\mathcal{G}) = \frac{\chi_\beta(\mathcal{G}; w) - \chi_\alpha(\mathcal{G}; w)}{\chi_\alpha(\mathcal{G}; w)}, \tag{4}$$

where $\chi_\alpha(\mathcal{G}; w)$ is the chromatic number, i.e, the smallest width required for coloring $\mathcal{G}$ under solution α, and $\chi_\beta(\mathcal{G}; w)$ is the chromatic number under solution β. Take the two solutions α and β of the graph shown in Figure 1 as an example, $\chi_\alpha(\mathcal{G}; w) = 28$, $\chi_\beta(\mathcal{G}; w) = 18$, therefore, the gap between α and β is $(18 - 28)/28 = -35.7\%$.

6.1 George and Appel's 27921 graphs

6.1.1 *AO-Graph coloring vs. memory coloring*

Table 2 shows that AOG Coloring obtains better solutions than Memory Coloring (with $ALIGN\_UNIT = 1$) in terms of the chromatic numbers of the colorings (allocations) in 98.64% of the 27921 IGs, and obtains worse solutions in only 0.29% of the IGs. When the $ALIGN\_UNIT$ used in Memory Coloring is increased to 2, AOG Coloring obtains better solutions in 99.23% IGs, and obtains worse solutions in only 0.17% IGs.

Let α be the solution found by AOG Coloring, and β be the solution from Memory Coloring. Table 3 shows the gaps between AOG Coloring and Memory Coloring calculated by (4). As the results shown, when $ALIGN\_UNIT = 1$ in Memory Coloring, the gaps are more than 20% on 68.31% graphs, with the maximum gap larger than 100%. When $ALIGN\_UNIT = 2$, the gaps are more than 20% on 69.13% graphs, and there are two graphs such that the gaps are larger than 100%.

If a graph is a comparability graph, its optimal interval coloring could be found in polynomial time [27]. And a graph could be recognized whether it is a comparability graph in polynomial time [27]. We implemented the comparability recognition algorithm, and found that there are total 1472 comparability graphs out of the 27921 graphs. We implemented the optimal interval coloring algorithm for comparability graphs to further test the optimality of our algorithm as well as Memory Coloring. The results are shown in Table 4. As the results shown, AOG Coloring obtains optimal solutions in 87.98% of the 1472 IGs. In contrast, the solutions from Memory Coloring are mostly sub-optimal, with only 5.43% of the 1472 IGs being optimal.

Table 5 Colorings found by Best-Fit compared to those found by First-Fit for 27921 IGs

Better	Equal	Worse
15933 (57.06%)	11016 (39.45%)	972 (3.48%)

Table 6 The gap between Best-Fit and First-Fit for 27921 IGs

$\leqslant -20\%$	$\leqslant -10\%$	$< 0\%$	0%	$> 0\%$	$\geqslant 10\%$	$\geqslant 20\%$	$\geqslant 30\%$	$\geqslant 40\%$	$\geqslant 50\%$	$\geqslant 60\%$	$\geqslant 70\%$
1	27	944	11016	8241	5168	1914	478	102	23	5	2

Table 7 Colorings found by Aggressive Simplify compared to two conservative simplification criterions for 27921 IGs

Criterion	Better	Equal	Worse
Conservative Simplify-1	7357 (26.35%)	15475 (55.42%)	5089 (18.23%)
Conservative Simplify-2	7054 (25.26%)	15653 (56.06%)	5214 (18.67%)

Table 8 The gap between Aggressive Simplify and Conservative Simplify-1/2 for 27921 IGs

Criterion	$\leqslant -20\%$	$\leqslant -10\%$	$< 0\%$	0%	$> 0\%$	$\geqslant 10\%$	$\geqslant 20\%$	$\geqslant 30\%$	$\geqslant 40\%$
Conservative Simplify-1	1	286	4802	15475	5498	1508	301	44	6
Conservative Simplify-2	5	288	4921	15653	5204	1511	300	36	3

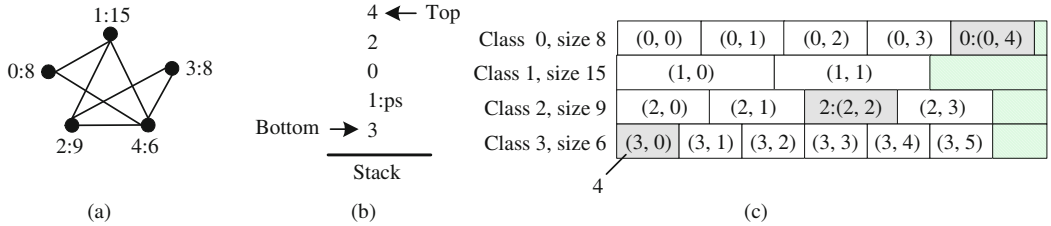

Figure 10 Another example IG and its memory coloring ($K = 41$). (a) IG (Graph #176 [1]); (b) node stack; (c) the pseudo register file and coloring.

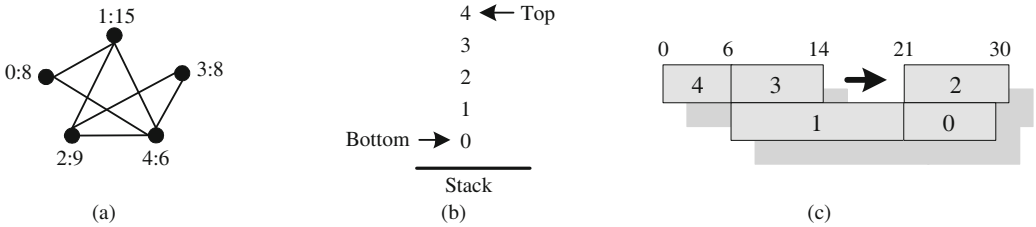

Figure 11 Another example IG and its AOG coloring (with Best-Fit) ($K = 30$). (a) IG (Graph #176 [1]); (b) node stack; (c) coloring.

6.1.2 *Best-Fit vs. First-Fit*

To compare the Best-Fit heuristics with First-Fit heuristics, we replace the Best-Fit heuristics in Select phase of our algorithm framework with First-Fit. Table 5 shows that Best-Fit obtains better solutions in 57.06% of the 27921 IGs, equal solutions in 39.45%, and obtains worse solutions in only 3.48% of the IGs.

Let α be the solution found by Best-Fit heuristics, and let β be the solution from First-Fit heuristics. Table 6 shows the gaps between Best-Fit and First-Fit calculated by (4). As the results show, the gaps are more than 10% on 27.54% graphs, with the maximum gap larger than 70%.

6.1.3 *Aggressive simplify vs. conservative simplify*

The goal of these experiments is to compare different simplification criterions. We implemented the conservative simplification heuristics in generalized graph coloring as well as Memory Coloring, called Conservative Simplify-1, which will yield a different coloring order with Aggressive Simplify developed in

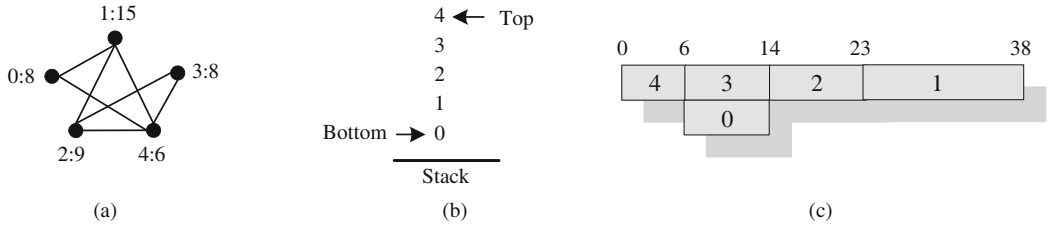

Figure 12 Another example IG and its AOG coloring (with First-Fit) ($K = 38$). (a) IG (Graph #176 [1]); (b) node stack; (c) coloring.

our algorithm.

Besides, we also developed another conservative simplification criterion in the context of interval coloring, which is based on Theorem 2.

Theorem 2. Suppose the weighted graph $(\mathcal{G}; w)$ contains a node x, such that $w(x) \leqslant \frac{k-w(N(x))}{|N(x)|+1}$, where K is the SPM size. Then $\mathcal{G}[V]$ is K-interval colorable if and only if $\mathcal{G}[V - \{x\}]$ is K-interval colorable.

Proof. For $x \in V$ such that $w(x) \leqslant \frac{k-w(N(x))}{|N(x)|+1}$, the number of gaps formed by x's neighbors is at most $|N(x)| + 1$ with a total width of at least $K - w(N(x))$; therefore, there must exist a gap such that its width is no smaller than $\frac{k-w(N(x))}{|N(x)|+1}$. Then x is always colorable.

In the context of register allocation, $w(x) = 1$, $w(N(x)) = |N(x)|$. The inequality is simplified into $|N(x)|+1 \leqslant k-|N(x)|$, namely $|N(x)| \leqslant \frac{k-1}{2}$. The traditional simplification criterion 1 is $|N(x)| \leqslant k-1$. Therefore, it is a conservative generalization but preserves the colorability. We call it Conservative Simplify-2 criterion. In the following experiments, we will compare Aggressive Simplify with the two conservative simplification criterions.

Table 7 shows the statistics of colorings found by Aggressive Simplify compared to the two conservative simplification criterions for 27921 IGs. As shown by the results, Aggressive Simplify is slightly better than the two conservative simplification criterions.

Let α be the solution found by Aggressive Simplify, and let β be the solution from Conservative Simplify-1 and Conservative Simplify-2. Table 8 shows the gaps between Aggressive Simplify and Conservative Simplify-1/2 calculated by (4).

6.1.4 *An example*

We demonstrate the advantages of our approach with another example IG, with the #176 IG from the George and Appel's 27921 IGs. Figure 10 shows the coloring result under memory coloring with $ALIGN\_UNIT = 1$. As shown, when $K = 41$, for node 1, no valid register is available for it. In fact, the minimum SPM size needed under memory coloring is 42 (1 unit larger than the SPM size given). In contrast, as shown in Figure 11, the minimum SPM size needed under our AOG coloring is only 30, with the gap being 40%. Figure 12 shows the result under First-Fit (using the same coloring order to AOG coloring). The minimum SPM size needed is 38. As a result, AOG coloring outperforms memory coloring and First-Fit.

6.2 DIMACS benchmarks

We also performed experiments on all the 73 weighted DIMACS benchmarks[2)]. Those DIMACS benchmarks are weighted graphs collected from real-life applications and graph theory researches. Let α be the solution found by AOG Coloring, and β be the solution from Memory Coloring ($ALIGN\_UNIT = 1$), Table 9 shows the gaps between AOG Coloring and Memory Coloring calculated by (4). As the results show, AOG Coloring obtains better colorings on all of the graphs, in which, the gaps are more than 20% on 83.56% graphs, with the maximum gap up to 122%.

Table 9 The gap between AO-Graph Coloring and Memory Coloring ($ALIGN\_UNIT = 1$) for 73 DIMACS benchmarks

Gap	Benchmarks
$> 0\%$	DSJC125.5g, GEOM30b, GEOM80b, GEOM100b, GEOM110b
$\geqslant 10\%$	GEOM20b, GEOM60b, GEOM120b, queen8_8g, queen12_12g, R75_5g, R100_9g
$\geqslant 20\%$	DSJC125.1g, queen10_10g, GEOM20, GEOM100, GEOM50b, GEOM60a, GEOM80b queen11_11g, R50_1gb, R50_5g, GEOM30a, GEOM40, GEOM70, GEOM80, GEOM90b
$\geqslant 30\%$	DSJC125.9g, GEOM30, GEOM40a, GEOM50, GEOM70a, GEOM70b, GEOM90, GEOM120 myciel6g, myciel7g, queen9_9g, queen10_10gb, queen11_11gb, R50_1g, R50_5gb R75_1g, R75_9g, R100_1g, R100_5g
$\geqslant 40\%$	GEOM50a, GEOM90a, GEOM110a, GEOM120a, queen9_9gb, R50_9g, R100_5gb
$\geqslant 50\%$	DSJC125.9gb, myciel5g, myciel5gb, queen12_12gb, GEOM20a, GEOM60 GEOM100a, GEOM110, R75_1gb, R75_9gb
$\geqslant 60\%$	DSJC125.5gb, GEOM80a, R50_9gb, R100_1gb, R100_9gb
$\geqslant 70\%$	DSJC125.1gb, myciel6gb, queen8_8gb, R75_5gb
$\geqslant 122\%$	myciel7gb

7 Conclusion

This paper proposes a new generalized graph coloring allocator for software-managed memory allocation. The novelty lies in incrementally constructing an acyclic orientation to the weighted IG, and integrating a new heuristics Best-Fit Select to choose color for nodes. This approach generalizes and subsumes as special cases the classical graph coloring register allocation algorithm without notably increased complexity. It deals with memory allocation while preserving the elegance and practicality of traditional graph coloring algorithm for register allocation. The effectiveness of our algorithm is validated by using a number of weighted IGs.

Acknowledgements

This work was supported in part by National Natural Science Foundation of China (Grant Nos. 61003081, 61370018), Funds for Creative Research Groups of China (Grant No. 60921062) and Australian Research Grants (Grant Nos. DP0987236, DP110104628).

References

1 Banakar R, Steinke S, Lee B-S, et al. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: Proceedings of the 10th International Symposium on Hardware/Software Codesign, Estes Park, 2002. 73–78

2 Dally W J, Labonte F, Das A, et al. Merrimac: supercomputing with streams. In: Proceedings of the ACM/IEEE Conference on Supercomputing, Phoenix, 2003. 35

3 Cuvillo J D, Zhu W, Ziang H U, et al. Fast: a functionally accurate simulation toolset for the cyclops64 cellular architecture. In: Proceedings of Workshop on Modeling, Benchmarking, and Simulation, Madison, 2005. 11–20

4 Makino J, Hiraki K, Inaba M. GRAPE-DR: 2-pflops massively-parallel computer with 512-core, 512-Gflops processor chips for scientific computing. In: Proceedings of the ACM/IEEE Conference on Supercomputing, New York, 2007. 1–11

5 Barker K J, Davis K, Hoisie A, et al. Entering the petaflop era: the architecture and performance of roadrunner. In: Proceedings of the ACM/IEEE Conference on Supercomputing, Austin, 2008. 1–11

6 Verma M, Wehmeyer L, Marwedel P. Dynamic overlay of scratchpad memory for energy minimization. In: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis, Stockholm, 2004. 104–109

7 Kandemir M, Ramanujam J, Irwin J, et al. Dynamic management of scratch-pad memory space. In: Proceedings of the 38th Conference on Design Automation, Las Vegas, 2001. 690–695

8 Li L, Gao L, Xue J L. Memory coloring: a compiler approach for scratchpad memory management. In: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Washington, 2005. 329–338

9 Li L, Nguyen Q-H, Xue J L. Scratchpad allocation for data aggregates in superperfect graphs. In: Proceedings of the

ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, San Diego, 2007. 207–216
10 Li L, Feng H, Xue J L. Compiler-directed scratchpad memory management via graph coloring. ACM Trans Archit Code Optim, 2009, 6: 1–17
11 Li L, Xue J L, Knoop J. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. ACM Trans Embed Comput Syst, 2011, 10: 28
12 Wan Q, Wu H, Xue J L. WCET-aware data selection and allocation for scratchpad memory. In: Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems, Beijing, 2012. 41–50
13 Chaitin G J. Register allocation & spilling via graph coloring. In: Proceedings of the SIGPLAN Symposium on Compiler Construction, Boston, 1982. 98–101
14 Chow F C, Hennessy J L. The priority-based coloring approach to register allocation. ACM Trans Program Lang Syst, 1990, 12: 501–536
15 Briggs P, Cooper K D, Torczon L. Improvements to graph coloring register allocation. ACM Trans Program Lang Syst, 1994, 16: 428–455
16 George L, Appel A W. Iterated register coalescing. ACM Trans Program Lang Syst, 1996, 18: 300–324
17 Smith M D, Ramsey N, Holloway G. A generalized algorithm for graph-coloring register allocation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Washington, 2004. 277–288
18 Wang L, Yang X J, Xue J L, et al. Optimizing scientific application loops on stream processors. In: Proceedings of the ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, Tucson, 2008. 161–170
19 Yang X J, Wang L, Xue J L, et al. Comparability graph coloring for optimizing utilization of stream register files in stream processors. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Raleigh, 2009. 111–120
20 Yang X J, Wang L, Xue J L, et al. Comparability graph coloring for optimizing utilization of software-managed stream register files for stream processors. ACM Trans Archit Code Optim, 2012, 9: 5
21 Fabri J. Automatic storage optimization. SIGPLAN Not, 1979, 14: 83–91
22 Garey M R, Johnson D S. Computers and Intractability: a Guide to the Theory of NP-Completeness. New York: W. H. Freeman & Co., 1979
23 Gergov J. Algorithms for compile-time memory optimization. In: Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, 1999. 907–908
24 Kierstead H A. A polynomial time approximation algorithm for dynamic storage allocation. Discrete Math, 1991, 87: 231–237
25 Buchsbaum A L, Karloff H, Kenyon C, et al. Opt versus load in dynamic storage allocation. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing, San Diego, 2003. 556–564
26 Lefebvre V, Feautrier P. Automatic storage management for parallel programs. Parallel Comput, 1998, 24: 649–671
27 Golumbic M C. Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57). Amsterdam: North-Holland Publishing Co., 2004
28 Squire M B. Generating the acyclic orientations of a graph. J Algorithms, 1998, 26: 275–290
29 Barbosa V C, Szwarcfiter J L. Generating all the acyclic orientations of an undirected graph. Inf Process Lett, 1999, 72: 71–74
30 Stanley R P. Acyclic orientations of graphs. Discrete Math, 1973, 5: 171–178
31 Linial N. Hard enumeration problems in geometry and combinatorics. SIAM J Algebr Discrete Methods, 1986, 7: 331–335
32 Hack S, Goos G. Optimal register allocation for ssa-form programs in polynomial time. Inf Process Lett, 2006, 98: 150–155