Scratchpad Allocation for Data Aggregates in Superperfect Graphs

Lian Li[*†] Quan Hoang Nguyen[*] Jingling Xue[*†]

Programming Languages and Compilers Group, School of Computer Science and Engineering, UNSW, Australia[*]
National ICT Australia[†]
{lianli,quanhn,jingling}@cse.unsw.edu.au

Abstract

Existing methods place data or code in scratchpad memory, i.e., SPM by either relying on heuristics or resorting to integer programming or mapping it to a graph coloring problem.

In this work, the SPM allocation problem is formulated as an interval coloring problem. The key observation is that in many embedded applications, arrays (including structs as a special case) are often related in the following way: For any two arrays, their live ranges are often such that one is either disjoint from or contains the other. As a result, array interference graphs are often superperfect graphs and optimal interval colorings for such array interference graphs are possible. This has led to the development of two new SPM allocation algorithms. While differing in whether live range splits and spills are done sequentially or together, both algorithms place arrays in SPM based on examining the cliques in an interference graph. In both cases, we guarantee optimally that all arrays in an interference graph can be placed in SPM if its size is no smaller than the clique number of the graph. In the case that the SPM is not large enough, we rely on heuristics to split or spill a live range until the graph is colorable. Our experiment results using embedded benchmarks show that our algorithms can outperform graph coloring when their interference graphs are superperfect or nearly so although graph coloring is admittedly more general and may also be effective to applications with arbitrary interference graphs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers optimization; B.3.2 [*Memory Structures*]: Design Styles—Primary memory

General Terms Algorithms, Performance

Keywords Scratchpad memory, SPM, graph coloring, interval coloring, superperfect graph, SPM allocation

1. Introduction

The effectiveness of memory hierarchy is critical to the performance of a computer system. To overcome the ever-widening gap between the processor speed and memory speed, fast on-chip SRAMs are used. An on-chip SRAM is usually configured as a hardware-managed cache, which works by relying on hardware to dynamically map data or instructions from off-chip memory. In embedded processors, the on-chip SRAM is frequently configured as a scratchpad memory (i.e., SPM).

The main difference between SPM and cache is that SPM does not have the complex tag-decoding logic that cache uses to support the dynamic mapping of data or instructions from off-chip memory. Therefore, it becomes more energy and cost efficient [3]. In addition, SPM is managed by software, which can often provide better time predictability, an important requirement in real-time systems. Given these advantages, SPM is widely used in embedded systems. In some high-end embedded processors such as ARM10E, ColdFire MCF5 and Analog Devices ADSP-TS201S, a portion of the on-chip SRAM is used as an SPM. In some low-end embedded processors such as RM7TDMI and TI TMS370CX7X, SPM has been used as an alternative to cache.

Effective utilization of SPM is critical for an SPM-based system. Research on automatic SPM allocation for data has focused on how to place the data that are frequently used in a program in SPM so as to maximize for both improved performance and energy consumption in the program [17, 23, 24, 21, 15, 2].

In [17], we introduced a compiler approach, called *memory coloring*, to automatically placing static data aggregates such as arrays and structs in SPM. Note that structs can be handled as a special case of arrays. Memory coloring is a general-purpose compiler approach that solves the SPM allocation problem by mapping it to a well-understood register allocation problem. The basic idea is to partition a given SPM into a pseudo register file, which consists of register classes for holding data aggregates of different sizes and then apply an existing graph coloring algorithm to color the data aggregates. To allow arrays to be dynamically swapped into and out of SPM, live range splitting is applied to some arrays to generate the data transfer statements required between SPM and off-chip memory. Like graph coloring register allocation, memory coloring is performed on the array interference graph of a program, which is built based on the liveness information for arrays. This liveness information is obtained by extending the liveness analysis for scalars to deal with the fact that an array may be live across function boundaries and accessed via aliased pointers. The effectiveness of graph coloring algorithms in register allocation and their scalability in handling large-scale programs make this approach promising.

By studying the array interference graphs of some embedded applications, we find that our general-purpose memory coloring approach can be further improved for a quite large class of embedded applications. The key observation is that in some embedded applications, two interfering arrays (i.e., two arrays connected by an edge in an interference graph) are often related in such a way that the live range of one array contains the live range of the other. An array live range A is said to *contain* an array live range B if A is live at every program point where B is live. In this case, two arrays are said to be *containing-related*.

Based on this observation, we propose a new approach that formulates the SPM allocation problem as an *interval coloring problem*. The interval coloring problem [12] is a generalization of the graph coloring problem to node weighted graphs.

Definition 1. *Given a node weighted graph $\mathcal{G} = (V, E)$ and positive-integral vertex weights $w = V \to N$, the interval coloring problem seeks to find an assignment of an interval I_u to each vertex $u \in V$ such that two constraints are satisfied: (1) for every vertex $u \in V$, $|I_u| = w_u$ and (2) for every pair of adjacent vertices u and v, $I_u \cap I_v = \emptyset$.*

The goal of interval coloring is to minimize the span of intervals $|\bigcup_v I_v|$ required in a valid coloring. Note that when every node in the graph has a unity weight, the interval coloring problem degenerates into the traditional graph coloring problem.

Let us recall below some standard definitions for a node weighted graph $\mathcal{G}$:

- A *clique* in $\mathcal{G}$ is a complete subgraph of $\mathcal{G}$. A clique is a *maximal clique* if it is not contained in any other cliques in $\mathcal{G}$. The *order* of a clique is the sum of the weights of all nodes in the clique.
- The *chromatic number* of $\mathcal{G}$ is the minimal span of intervals needed to color $\mathcal{G}$.
- The *clique number* of $\mathcal{G}$ is the order of a largest maximal clique in $\mathcal{G}$.

In general, the chromatic number of a node weighted graph is equal to or greater than its clique number. In the special case when the chromatic number of a graph is equal to its clique number, the graph is known as a *superperfect* graph. (A superperfect graph is known as a *perfect graph* if all its nodes have unity weights.)

The SPM allocation problem can be naturally abstracted as an interval-coloring problem as follows. Each node, i.e., array object in an array interference graph is weighted with its corresponding size. Allocating SPM spaces to array live ranges is represented by the assignment of intervals to vertices of the graph. Minimizing the span of intervals amounts to minimizing the required SPM size.

Interval coloring is an NP problem and how to recognize and color a superperfect graph remains to be an open problem [12]. In this paper, however, we show that an array interference graph is a superperfect graph if any pair of array live ranges in the graph are either disjoint or containing-related. Furthermore, we will present two algorithms that each can find an optimal allocation for such an array interference graph in polynomial time when the size of a given SPM is equal to the clique number of the graph. If the SPM is not large enough, both algorithms use heuristics to split or spill some arrays until an optimal SPM allocation for the resulting interference graph is possible. The two algorithms differ in their live range splitting strategies used. One performs live range splitting before spilling while the other performs live range splitting together with spilling. In both algorithms, we use a simple cost model to evaluate the colorability of an array interference graph and guide spilling and splitting decisions. Our experimental results using a set of benchmarks from MediaBench and EEMBC show that our algorithms are effective for these embedded applications and can outperform the memory coloring approach in most benchmarks.

In summary, this paper makes the following contributions:

- We uncover and study some salient properties about array live ranges in embedded applications.
- We propose a new methodology for solving the SPM allocation problem by formulating it as an interval coloring problem.
- We present a new cost model to guide splitting and spilling and two interval-coloring algorithms for placing arrays in SPM.
- We have implemented this work in SUIF and MachSUIF and evaluated its effectiveness using a set of embedded benchmarks.

The rest of the paper is organized as follows. Section 2 introduces some definitions about array live ranges and discuss a live range splitting heuristic used in this paper. In Section 3, we uncover and study some salient properties about array live ranges in embedded applications. Sections 4 and 5 present our two interval-coloring algorithms. We evaluate their effectiveness in Section 6. Section 7 reviews some related work and Section 8 summarizes this paper.

2. Preliminary: Array Live Ranges

The *live range* of an array in a program is the union of all the program points in the control flow graph of the program that link the different definitions of this particular array to their uses. An array is said to be *live* at a program point if this point belongs to its live range. As for scalars, *live range splitting* can also be applied to arrays by inserting array copy statements so that an array can be split into several arrays with smaller live ranges.

2.1 Live Range Splitting

This is generally useful since arrays often have long live ranges and may be frequently accessed only in parts of their live ranges. By splitting an array so that its frequently accessed parts become distinct new arrays with shorter live ranges, we can increase the chances for the shorter live ranges to be placed in SPM.

In this paper, we use the same live range splitting strategy described in [17] to split arrays at hot loops (including call sites as a special kind of loops) since most array accesses come from inside loops. We have improved our earlier strategy by allowing an array to be split even if it may be accessed indirectly by a pointer that may also point to other arrays. This is implemented by using runtime method tests that are often used for devirtualizing or inlining virtual function calls in object-oriented programs [6].

As in [17], a simple cost mode is used to decide if the live range of an array A in a loop L should be split into a new array A' based on the (optimistic) assumption that A' will end up being kept in SPM if the splitting is performed. Due to live range splitting, an array copy statement $A' = A$ is introduced at the pre-header of L and $A = A'$ at an exit of L if A may be modified inside and live at the exit. All accesses to A (including those accessed indirectly by pointers) in L will be replaced by those to A'. So the cost of splitting A at L is $(C_s + C_t \times A.size) \times numcopies$, where C_s is the startup cost, C_t is the transfer cost per byte, $A.size$ is the size of A (in bytes) and $numcopies$ is 1 or 2 depending on whether $A = A'$ is executed or not. The benefit for the splitting is $A.freq \times (M_{mem} - M_{spm})$, where $A.freq$ is the access frequency of A in L, M_{mem} is the memory access latency (in cycles) and M_{spm} is the SPM access latency (in cycles).

If the benefit is larger than the cost for a particular split, the split will be performed. Every array that is accessed in a loop nest will be split at most once in one of its loops when its loops are processed outside in. All new arrays introduced inside loops are called *hot arrays*; these loops are called *hot loops*. All the arrays that appear originally in the program are called *nonsplit arrays*.

2.2 Liveness Analysis

The liveness information for the arrays in a program is required in order for us to build the interference graph for these arrays. Two arrays are *interfering* with each other if the live range of one array contains a definition of the other array. The interfering arrays cannot be kept in overlapping SPM spaces.

Liveness analysis is performed after live range splitting has been done. Our approach to the SPM allocation problem is interprocedural in the sense that all arrays in a program are considered for SPM allocation at the same time. Many embedded applications are free of recursion. So a caller/callee save and restore mechanism may be dispensed with. As a result, we use the same liveness analysis for arrays described in [17] to compute the liveness information for all arrays in a program inter-procedurally. In particular, an array is considered to be live on entry to a call site if it is live on entry to a callee function that may be invoked from the call site. This is necessary when the array is global and used in the callee function or when the array (global or local) is passed by reference to the callee function and accessed indirectly in the callee function. In addition, due to the absence of a caller-callee save mechanism for arrays, an array that is live at the exit of of a call site is assumed to be live on entry of every function that may be invoked from the call site.

3. Superperfectness

In this section, we provide evidence to show that in many embedded applications, two arrays are often containing-related whenever they interfere. We will show further that an interference graph is a superperfect graph if any pair of arrays in the interference graph are either disjoint or containing-related. Finally, we discuss briefly the basic idea about coloring such a superperfect graph.

3.1 Containment

As illustrated in Figure 1, two array live ranges can be related in one of the three ways. Figure 1(a) gives an example of containing-related arrays. In Figure 1(b), the two arrays are disjoint and thus not interfering. In Figure 1(c), the two arrays interfere with each other but neither live range contains the other. However, in almost all embedded applications we have studied, the situation in Figure 1(c) happens rarely. Frequently, Property 1 holds for two arrays.

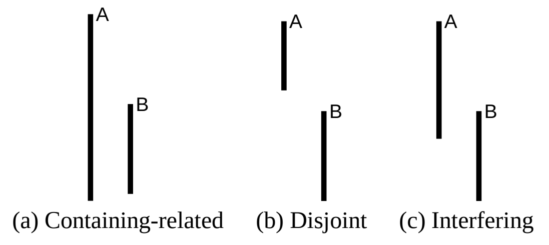


Figure 1. Containing-related, disjoint and interfering arrays.

Property 1. *If two arrays in a program interfere, then the live range of one array contains that of the other.*

As a result, any two arrays in a program are either disjoint or containing-related. We draw this conclusion based on the observation from a number of representative embedded benchmarks.

In this paper, a *definition block* is referred to a scope, e.g., a compound statement in C, where arrays are declared. All nonsplit arrays in a program are conservatively assumed to be defined only once at their respective definition blocks. All global arrays are treated as a special case; they are considered to be defined in the outermost scope in a pseudo function that is the only caller of the main function of a program. A hot array is assumed to be defined at the entry of its associated hot loop even though it its initialization statement appears in the pre-header of the loop.

In this work, we will consider to place hot and nonsplit arrays in SPM as follows. If an array live range A is split into several hot arrays, say, $A_1, A_2, \ldots, A_n$, then only one of the following two scenarios will occur. In one scenario, all these hot arrays are considered for SPM allocation and the remaining no-hot portions of the live range are ignored. These non-hot pieces will reside in off-chip memory since they are infrequently accessed. This is reasonable since the majority of array accesses to A will be from those to the hot parts of its live range, $A_1, A_2, \ldots, A_n$. In the other scenario, A is considered for SPM allocation and all its hot live ranges represented by the hot arrays $A_1, A_2, \ldots, A_n$ are ignored (as if no live range splitting for A has ever taken place).

To ensure that Property 1 holds for a program, the following three assumptions about the live ranges of the hot and nonsplit arrays in the program are made.

Assumption 1 If an array is live on entry to a definition block, then it is live in the entire scope of the definition block.

Assumption 2 If an array is live on entry to a call site, then it is live at the exit of the call site.

Assumption 3 For two arrays defined in the same definition block, every last use of one array must be post-dominated by at least one last use of the other array.

We have studied the array live range behavior in a set of representative embedded applications from MediaBench and EEMBC benchmark suites (cf. Table 1). Only four arrays in pegwitencode and pegwitdecode do not satisfy these three assumptions.

Property 1 holds for a program if these three assumptions are all satisfied. Below we present further explanations.

Let us first examine Assumption 1, which is relevant to the arrays defined in different scopes. We assume that scopes are either nested or disjoint as in C. According to this assumption, every live range in a scope must contain all live ranges inside its nested scopes. This assumption seems to be restrictive. But there does not appear to be a need to relax it since the local arrays in a function are usually declared in its outermost scope in embedded applications.

Assumption 2 takes care of arrays defined in different functions. For two arrays defined in two different functions, the array defined in the caller function must be live on entry to the callee function if they interfere with each other. According to this assumption, the array in the caller is also live at the exit of the call site. Therefore, it is live through the callee function and contains the live ranges of all arrays defined in the callee function.

Finally, let us examine Assumption 3, which is applicable to all arrays defined in a common definition block. Consider two such arrays, A and B. Initially, both are regarded conservatively as being defined only once at the beginning of the block. According to this assumption, A contains B if every last use of B is post-dominated by some last use of A. Similarly, the converse is true.

3.2 Superperfect Graphs

We show that if Property 1 holds for every pair of interfering arrays in an interference graph, then all arrays in the graph can be colored if the size of a given SPM is equal to the clique number of the graph. This implies that the interference graph under consideration is a superperfect graph. Thus an optimal interval coloring is achieved.

Theorem 1. *If Property 1 is true for all arrays in a program, then the interference graph for the program is a superperfect graph.*

The proof of this theorem follows from two lemmas stated below. Let a *containment non-decreasing order* be a total order that is defined on the set of all live ranges in a program as follows. Suppose A and B are two arbitrary live ranges. If A and B are disjoint or identical (i.e., contain each other), then either A precedes B or A succeeds B. Otherwise, A and B must be containing-related. Then the contained live range precedes the containing one.

Let $\mathcal{G}$ be an interference graph such that Property 1 holds for all live ranges in $\mathcal{G}$. Let all arrays in $\mathcal{G}$ be colored in any given containment non-decreasing order. Suppose A_n is an arbitrary but fixed array in $\mathcal{G}$. Let $\mathcal{G}_{A_n}$ be a subgraph of $\mathcal{G}$ including all arrays that are colored before A_n. Let $\mathcal{C}_{A_n} = \{A_0, A_1, \ldots, A_{n-1}\}$ be a clique in $\mathcal{G}_{A_n}$ with the largest order such that all arrays in $\mathcal{C}_{A_n}$ interfere with A_n and are colored from A_0 to A_{n-1} in the given containment non-decreasing order. We say that A_n is *perfectly placed* in SPM if it is placed in SPM immediately after A_{n-1} from the lowest to highest address in the SPM. By convention, A_0 is placed at the offset 0 in SPM, in which case $\mathcal{C}_{A_0} = \emptyset$.

Lemma 1. *An array A can be perfectly placed in SPM with an infinite size if all live ranges in $\mathcal{G}_A$ are perfectly placed.*

Proof. We will prove this case by contradiction. Suppose A_n cannot be placed in SPM immediately after A_{n-1}. Then there must exist an array Z_m in $\mathcal{G}_{A_n}$ that interferes with A_n. Furthermore, since Z_m has been perfectly placed, it is always possible to choose Z_m so that the order of the clique $\{Z_m\} \cup \mathcal{C}_{Z_m} = \{Z_0, Z_1, \ldots, Z_{m-1}, Z_m\}$ is larger than that of $\mathcal{C}_{A_n}$. This is possible since otherwise A_n can already be perfectly placed. This contradicts the fact that $\mathcal{C}_{A_n}$ has the largest order for A_n. □

Lemma 2. *$\mathcal{G}$ can be colored with an SPM size larger than or equal to its clique number if all live ranges in $\mathcal{G}$ are perfectly placed.*

Proof. By Lemma 1, every array A_n in $\mathcal{G}$ can be kept in SPM if the SPM is larger than or equal to the order of $\{A_n\} \cup \mathcal{C}_{A_n} = \{A_0, A_1, \ldots, A_{n-1}, A_n\}$. Furthermore, the order of $\{A_n\} \cup \mathcal{C}_{A_n}$ for some A_n must be the clique number of the graph. □

3.3 Methodology

As discussed earlier, we will allocate SPM spaces only to the hot and nonsplit arrays in a program. Property 1 holds for most of these arrays in the embedded benchmarks that we have studied. Consequently, their interference graphs are often superperfect graphs.

For a given program, we will start with an interference graph including only its hot and nonsplit arrays. Such an interference graph is very likely to be superperfect. If it is not, we will extend the live ranges of some interfering arrays so that any two interfering arrays are either disjoint or containing-related. This a safe and conservative approximation of the liveness information for arrays in a program. For the set of 10 embedded applications we have studied (cf. Table 1), only four live ranges in the two benchmarks, pegwitencode and pegwitdecode, need to be extended.

According to Theorem 1, the resulting interference graph for a program is superperfect and can thus be optimally colored if the size of a given SPM is no smaller than the clique number of the graph. If the SPM is smaller (which may often be the case in practice), some heuristics are applied to split or spill some live ranges until the resulting interference graph becomes optimally colorable. In this case, all live ranges that remain in the interference graph can be optimally placed in SPM. Two different implementations of this methodology are developed and presented below. They differ in whether live range splitting and spilling are done sequentially (*aggressive splitting*) or together (*on-demand splitting*).

When each algorithm is presented, $\mathcal{G}$ consistently denotes the interference graph being processed by the algorithm.

3.4 An Example

Our illustrating example is given in Figure 2(a). The hot loops BB2, BB3, BB6 and BB7 are highlighted in grey. The program has four arrays A, B, C and D. Their sizes are 80, 240, 80 and 80 bytes, respectively. A and B are defined in the main function and are live through the whole program. C and D are defined in functions g and f, respectively. Function f is a callee function invoked at a call site in function g. As a result, A and B contain each other. Both A and B contain C and D. D is contained by the other three arrays.

The live ranges of all arrays after live range splitting has been performed are illustrated graphically in Figure 2(b). The hot arrays A1, B1, C1 and D1 are introduced in the hot loops (and call sites) as highlighted. Their live ranges are contained by the live ranges of the nonsplit, i.e., original arrays A and B. The live ranges of A1 and B1 are disjoint. A1 is live inside the functions g and f. Thus, the live range of A1 contains all live ranges defined in g and f, including the nonsplit arrays C and D as well as the hot arrays C1 and D1. Similarly, B1 contains D and D1.

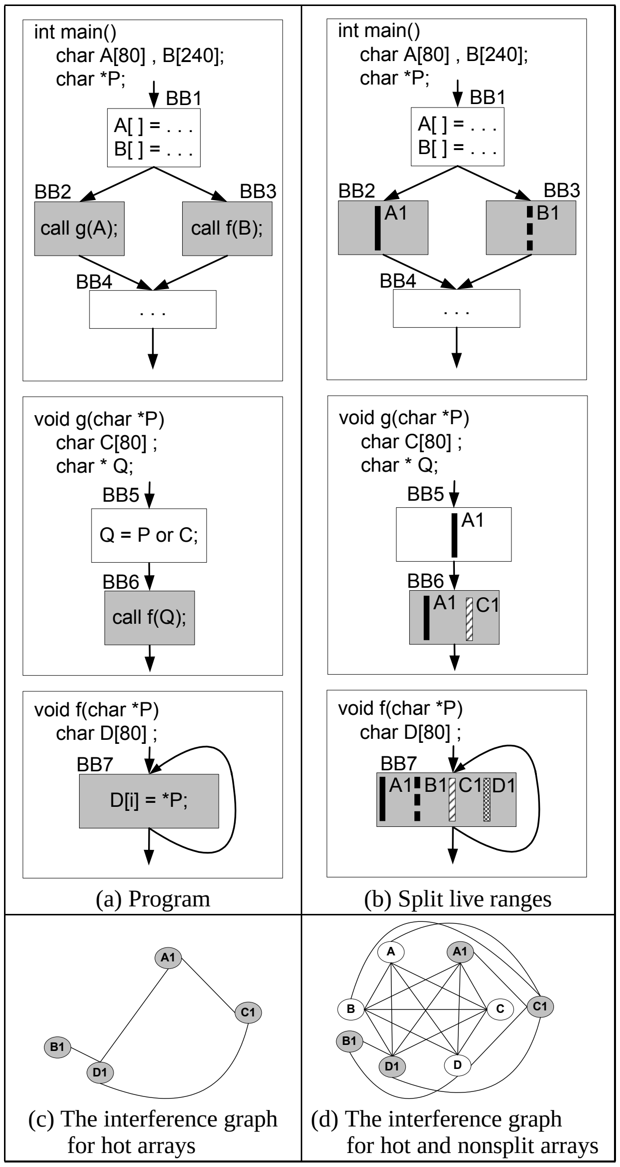


Figure 2. A motivating example for aggressive splitting.

4. Aggressive Splitting

Figure 3 gives the phase ordering of our aggressive splitting framework. It starts with an interference graph including only the hot arrays in a program. When the interference graph becomes colorable, the nonsplit arrays in the program are gradually included as long as the resulting interference graph is still colorable.

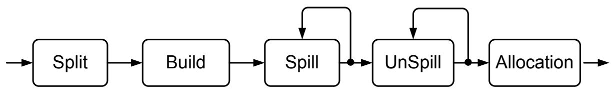


Figure 3. Aggressive splitting.

4.1 Build

Two tasks are performed for a program. First, an interference graph including only the hot arrays in the program is built. Second, all nonsplit arrays in the program are pushed into a so-called $unspill\_queue$ where they will be examined in the Unspill phase.

For the motivating example, the interference graph for the hot arrays is depicted in Figure 2(c). The $unspill\_queue$ is initialized to contain the four original arrays A, B, C and D in the example.

Algorithm 1 An algorithm for finding all maximal cliques in an array interference graph $\mathcal{G}$ for a program.

```
 1: procedure FIND_ALL_CLIQUES
    // Find all maximal cliques in the interference graph $\mathcal{G}$
 2:     for every function $f$ in the call graph of the program in reverse topological order do
 3:         Find_Func_Cliques($f$)
 4:     end for
 5: end procedure

 6: procedure FIND_FUNC_CLIQUES($f$)
    // Find maximal cliques of function $f$, denoted $f.cliqueset$
 7:     Let $S$ be the set of arrays in the given interference graph $\mathcal{G}$
 8:     $f.S = \{f$'s local arrays$\} \bigcap S$
 9:     for every program point $P$ in $f$ do
10:         $localclique = \{l \mid l \in live(P) \wedge l \in f.S\}$
11:         if $P$ is an entry point to a call site then
12:             Let $gset$ be the set of all callee functions that may be invoked at the call site
13:             for every $g \in gset$ such that $g.cliqueset \neq \emptyset$ do
14:                 Add to $localcliqueset$ the set of cliques formed by combining $localclique$ with every clique in $g.cliqueset$
15:             end for
16:         else
17:             $localcliqueset = \{localclique\}$
18:         end if
19:         $f.cliqueset \bigcup= localcliqueset$
20:     end for
21: end procedure
```

The procedure Find_All_Cliques given in Algorithm 1 is applied to find all maximal cliques in an array interference graph for a program. The algorithm works in a bottom-up manner by traversing the call graph of the program in reverse topological order (line 2). For simplicity, every definition block where some arrays are defined is regarded as a special inlined function called from its immediately enclosing outer scope. In addition, a pseudo function is introduced to represent the outermost scope where all global arrays are defined. In the call graph for a program, this pseudo function is the only caller function for the main function in the program. When visiting a function f, the procedure Find_Func_Cliques is called to update the set of cliques that is already identified by incorporating the local arrays defined in f. The set of cliques after f's local arrays have been included is represented as $f.cliqueset$ and will be used when the caller functions of f are processed.

In Find_Func_Cliques, every program point P in f is visited (line 9). The local arrays of f that are live simultaneously at P form a clique (line 10), denoted $localclique$. If an array is live at a call site, then it interferes with every array defined in any callee function that may be invoked at the call site. Thus, on entry to each call site, the set of cliques grows by including those obtained by combining $localclique$ with each clique in the clique set found in every callee function invokeable at the call site (lines 11 - 15).

After the pseudo function, where all the global arrays are conceptually defined, has been visited, all arrays in the program will have been processed. All maximal cliques in the given interference graph for the program have thus been identified.

Consider the example program given in Figure 2(a). Since only the hot arrays are considered, S = {A1,B1,C1,D1}. As the only leaf function in the program, f is visited first and the clique that consists of the singleton hot array D1 is included in $f.cliqueset$. Next, g is visited. At BB6, $localclique$ = {C1} is combined with each clique in $f.cliqueset$. As a result, $g.cliqueset$ = {{D1,C1}}. After main has been processed, two maximal cliques {A1,C1,D1} and {B1,D1} are identified finally. Due to the absence of global arrays, the largest maximal clique in this interference graph is {B1,D1} with an order of 320 bytes. So the clique number and chromatic number of this interference graph are both 320 bytes.

4.2 Spill

This phase reduces the clique number of an uncolorable interference graph by spilling some arrays. We spill an array by using a new cost model developed specifically for superperfect graphs. Spilling an array means removing it from the interference graph. All arrays remaining in the interference graph will verify Property 1. So this phase preserves the perfectness of the interference graph.

4.2.1 The Cost Model

A term *colorability* is introduced to represent the number of array accesses that can hit in SPM after SPM allocation. We use a simple cost model to evaluate the *colorability* of an interference graph.

For a clique $\mathcal{C}$, all arrays in $\mathcal{C}$ can be colored if the order of $\mathcal{C}$ is no larger than the given SPM size. Then the colorability of $\mathcal{C}$ is the sum of the frequencies of all arrays in $\mathcal{C}$. If $\mathcal{C}$ is not colorable, we approximate the colorability of $\mathcal{C}$ as $\alpha(\mathcal{C})$:

$$\alpha(\mathcal{C}) = \texttt{SPM\_SIZE} \times \frac{\mathcal{C}.freq}{\mathcal{C}.order}$$

where SPM_SIZE is the size of a given SPM, $\mathcal{C}.freq$ is the sum of the frequencies of all arrays in $\mathcal{C}$ and $\mathcal{C}.order$ is the order of $\mathcal{C}$.

For an interference graph $\mathcal{G}$, we estimate its colorability as the sum of the colorability quantities of all its maximal cliques:

$$\alpha(\mathcal{G}) = \sum_{\mathcal{C} \text{ is a maximal clique in } \mathcal{G}} \alpha(\mathcal{C})$$

In our cost model, the larger the colorability is for an interference graph, the better the allocation results will be.

4.2.2 The Algorithm

Algorithm 2 An algorithm for spilling arrays in aggressive splitting.

```
 1: procedure SPILL
 2:     Let $Cset$ be the set of maximal cliques with orders larger than SPM_SIZE in the given interference graph $\mathcal{G}$
 3:     while $Cset \neq \emptyset$ do
 4:         for every array $A$ in a clique from $Cset$ do
 5:             $A.spillpenalty = A.freq \times (M_{mem} - M_{spm})$
 6:             $A.spillbenefit = (\alpha(\mathcal{G} - A) - \alpha(\mathcal{G})) \times (M_{mem} - M_{spm})$
 7:             $A.spillcost = A.spillpenalty - A.spillbenefit$
 8:         end for
 9:         Select an array $A$ in a clique from $Cset$ with the least $A.spillcost$
10:         Push $A$ into $unspill\_queue$
11:         Remove $A$ from $\mathcal{G}$
12:         for every clique $\mathcal{C}$ that includes $A$ do
13:             Remove $A$ from $\mathcal{C}$
14:             if $\mathcal{C}.order \leq$ SPM_SIZE then
15:                 $Cset = Cset - \mathcal{C}$
16:             end if
17:         end for
18:     end while
19: end procedure
```

Algorithm 2 explains this phase in details. The performance loss of a single access from off-chip memory is estimated by ($M_{mem} - M_{spm}$), where M_{spm} and M_{mem} denote the cycle counts of one SPM access and one off-chip memory access, respectively. Therefore, the penalty incurred for spilling an array A (denoted $A.spillpenalty$) is estimated as a product of the access frequency of A and the performance loss of a single access from off-chip memory (line 5). The benefit of spilling A (denoted $A.spillbenefit$) is approximated according to the increased colorability of the interference graph after A has been removed (line 6). The overall cost of spilling A (denoted $A.spillcost$) is equal to the penalty $A.spillcost$ minus the benefit $A.spillbenefit$ (line 7).

We always choose to spill an array with the minimal overall spilling cost. The selected array, instead of being spilled immediately, is pushed into $unspill\_queue$ and will be examined in the Unspill phase (line 10). The interference graph and its maximal cliques are updated after an array has been spilled (lines 11 - 17). This phase terminates when the interference graph is colorable.

Let us be given an SPM of 320 bytes. For the program in Figure 2(a), the interference graph including only the hot arrays is given in Figure 2(c). Its clique number is exactly the same as the given SPM size. No spilling is required. So all the hot arrays can be placed in the SPM (as also suggested by Theorem 1).

4.3 Unspill

At this stage, the interference graph is optimally colorable. But the SPM may be under-utilized. In addition, it may be more beneficial to coalesce unnecessary splits introduced due to live range splitting. So this phase is introduced to overcome these two problems although its coalescing role is more significant one of the two.

All arrays in $unspill\_queue$, including both hot and nonsplit arrays, are examined. An array will be unspilled if it can be included in the interference graph without making it uncolorable. Unspilling a nonsplit array means that the nonsplit array is included in the interference graph with all hot arrays that are split from it being discarded. Then all the array copies introduced for splitting this array can be eliminated. Since the interference graph contains either a nonsplit array or its hot arrays split from it but not both, the resulting interference graph will remain to be superperfect (Theorem 1).

Algorithm 3 An algorithm for unspilling arrays

```
1: procedure UNSPILL
2:     Let $\mathcal{G}$ be the given interference graph
3:     for every array $A \in unspill\_queue$ do
4:         if $A$ is a nonsplit array that has been split then
5:             $A.unspillfreq$ is the frequency of $A$ minus the frequencies of all hot arrays in $\mathcal{G}$ that are split from $A$
6:             $A.copycost$ = the copy cost for copying all hot arrays in $\mathcal{G}$ between SPM and off-chip memory
7:             $A.unspillbenefit = A.unspillfreq \times (M_{mem} - M_{spm}) + A.copycost$
8:         else
9:             $A.unspillbenefit = A.freq \times (M_{mem} - M_{spm})$
10:        end if
11:    end for
12:    while $unspill\_queue \neq \emptyset$ do
13:        Select $A$ in $unspill\_queue$ with the largest $A.unspillbenefit$
14:        if $\mathcal{G}$ remains colorable with $A$ being included then
15:            Add $A$ into $\mathcal{G}$
16:            Update the maximal cliques in $\mathcal{G}$
17:        end if
18:        Remove $A$ from $unspill\_queue$
19:    end while
20: end procedure
```

The benefit of unspilling an array is computed according to the increased SPM accesses and the number of eliminated array copies by placing this array (rather than its split hot arrays) in SPM (lines 4 - 10). All arrays in $unspill\_queue$ are examined in the order from the largest to smallest unspilling benefit (line 13). If an array can be colored (line 14), then it will be included in the interference graph (line 15). Otherwise it will be accessed from off-chip memory. After an array has been unspilled, the interference graph and its maximal cliques are updated accordingly (lines 15 and 16).

For the example program, let us continue to assume an SPM of 320 bytes. A and B cannot be unspilled since it will result in the oversized cliques {A,B1,D1} and {A1,B,C1,D1}, respectively. However, C and D will be unspilled in that order. In the resulting interference graph, there are two cliques {A1,C,D} and {B1,D}.

4.4 Allocation

Algorithm 4 An algorithm for SPM allocation.

```
1: procedure COLOR_CLIQUE
   // Color all arrays remaining in the interference graph
2:     Let $L$ be the list of all arrays in the interference graph $\mathcal{G}$
3:     Sort $L$ in a containment non-decreasing order
4:     while $L \neq \emptyset$ do
5:         Remove the first array $A$ from $L$
6:         $availaddr = 0$
7:         for every array $B$ that has been colored do
8:             if $B$ interferes with $A$ and $B.address + B.size > availaddr$ then
9:                 $availaddr = B.address + B.size$
10:            end if
11:        end for
12:        $A.address = availaddr$
13:    end while
14: end procedure
```

Finally, all arrays remaining in the interference graph can be successfully placed in SPM. We present an algorithm that colors these arrays in a containment non-decreasing order.

As shown in Algorithm 4, all arrays are colored in a containment non-decreasing order (lines 2 and 3). An array will be placed in SPM at the first available SPM address (lines 5 - 12), which can be obtained by examining all interfering arrays that are already placed in SPM. By Theorem 1, all arrays can be successfully colored.

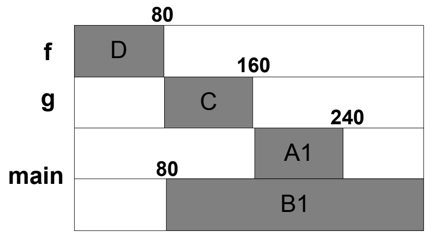


Figure 4. SPM allocation result for the example in Figure 2(a).

Consider the example in Figure 2(a). After Unspill, the interference graph contains four arrays: A1, B1, C and D. Figure 4 depicts the allocation result. There are two possible containment non-decreasing orders: (1) D, C, $A1$ and $B1$ and (2) D, C, $B1$ and $A1$. Let us assume that the former order is used. D is colored first and placed in SPM at the offset 0. Then C is placed at the offset 80. A1 will be placed in SPM at the offset 160 since it interferes with C and D. Finally, B1 can be placed in SPM at the offset 80. For this example, the same allocation will result if the other containment non-decreasing order is used.

5. On-demand Splitting

Unlike aggressive splitting, on-demand splitting starts with a larger interference graph that includes all nonsplit arrays, i.e., all original arrays in a program. If the interference graph is uncolorable, its clique number will be gradually reduced by splitting or spilling some arrays. The motivation behind is that some frequently accessed arrays may not be split successfully. In aggressive splitting, those arrays are considered only after all hot arrays can be colored. Therefore, they may not have a chance to be placed in SPM. This problem can be avoided in the on-demand splitting framework, where nonsplit arrays are split into hot arrays in an on-demand manner. The same cost model introduced in Section 4.2.1 is used to estimate the costs incurred for splitting and spilling an array.

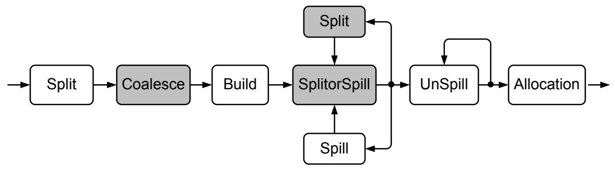


Figure 5. On-demand splitting.

Figure 5 depicts the phase ordering of the on-demand splitting framework. The phases that are different from the aggressive splitting framework are highlighted in grey and described below.

5.1 Coalesce

All copy-related arrays are coalesced. We introduce a flag *coalesced* for each array to denote whether it has been coalesced or not. The flag is initialized to false for all arrays. After coalescing, the flag is true for all hot arrays introduced in live range splitting.

5.2 Build

As in aggressive splitting, the same two tasks but with different semantics are performed. First, the interference graph constructed will include not only hot but also nonsplit arrays in the program. Second, $unspill\_queue$ will start being empty since all arrays for SPM allocation have been included in the interference graph.

The same procedure (Algorithm 1) used in aggressive splitting is applied to find the maximal cliques in the interference graph. The maximal cliques found may consist of both hot and nonsplit arrays. All arrays flagged as *coalesced* are not counted in computing the order of a clique. The reason in building an interference graph this way is to enable us to update the interference graph and its maximal cliques on the fly after an array has been split.

Figure 2(d) gives the interference graph for our motivating example, where the four coalesced nodes, A1, B1, C1 and D1, are highlighted. The two maximal cliques in this interference graph are {A,A1,B,C,C1,D,D1} and {A,B,B1,D,D1}. Note that the hot arrays A1, B1, C1 and D1 are flagged as *coalesced* and they will not be counted in computing the order and colorability of a clique.

5.3 SplitorSpill

When an interference graph is not colorable, we will split or spill an array in the graph to reduce the clique number of the graph. The cost and benefit of splitting and spilling an array are estimated based on the same cost model used in aggressive splitting.

In Algorithm 5, we estimate the cost and benefit of spilling an array A exactly as in aggressive splitting (lines 5 - 7).

Similarly, the overall cost of splitting A is the penalty incurred for splitting A minus the benefit (line 11). After A has been split, array copy statements are inserted. Only the hot arrays that are split from A are considered for coloring and those no-hot portions of the live range of A will be accessed from off-chip memory. Therefore,

Algorithm 5 An algorithm for splitting and spilling live ranges in on-demand splitting.

```
1:  procedure SPLITORSPILL
2:      Let $Cset$ be the set of maximal cliques with orders larger
        than SPM_SIZE in the given interference graph $\mathcal{G}$
3:      while $Cset \neq \emptyset$ do
4:          for every array $A$ in a clique from $Cset$ do
5:              $A.spillpenalty = A.freq \times (M_{mem} - M_{spm})$
6:              $A.spillbenefit = (\alpha(\mathcal{G}-A)-\alpha(\mathcal{G}))\times(M_{mem} - M_{spm})$
7:              $A.spillcost = A.spillpenalty - A.spillbenefit$
8:              $A.splitpenalty$ is the cost of splitting $A$
9:              Let $\mathcal{G}'$ be the modified interference graph after
                splitting $A$
10:             $A.splitbenefit = (\alpha(\mathcal{G}') - \alpha(\mathcal{G})) \times (M_{mem} - M_{spm})$
11:             $A.splitcost = A.splitpenalty - A.splitbenefit$
12:         end for
13:         Select $A$ in a clique from $Cset$ with the least
            $min(A.spillcost, A.splitcost)$
14:         Push $A$ into $unspill\_queue$
15:         Remove $A$ from $\mathcal{G}$
16:         if $A.spillcost \leq A.splitcost$ then
17:             for every clique $\mathcal{C}$ that includes $A$ do
18:                 Remove $A$ from $\mathcal{C}$
19:                 if $\mathcal{C}.order \leq$ SPM_SIZE then
20:                     $Cset = Cset - \mathcal{C}$
21:                 end if
22:             end for
23:         else
24:             for every clique $\mathcal{C}$ that includes $A$ do
25:                 Remove $A$ from $\mathcal{C}$
26:                 Set the flag $coalesced$ to false for the live
                    ranges split from $A$ in $\mathcal{C}$ (if any)
27:                 if $\mathcal{C}.order \leq$ SPM_SIZE then
28:                     $Cset = Cset - \mathcal{C}$
29:                 end if
30:             end for
31:         end if
32:     end while
33: end procedure
```

the penalty incurred for splitting A is the copy cost for the inserted copy statements plus the penalty incurred for accessing those no-hot portions of A's live range from off-chip memory (line 8). The benefit of splitting A is computed as the increased colorability of the interference graph due to the splitting (lines 9 and 10).

When splitting or spilling A, A is removed from the interference graph and pushed into $unspill\_queue$ (lines 14 and 15). The maximal cliques in the interference graph are updated accordingly (lines 16 - 31). If A is selected for splitting, the flag *coalesced* is set to false for every hot array that is split from A (line 26).

Let us continue to study the interference graph given in Figure 2(d). As in aggressive splitting, the SPM size is 320 bytes. The initial interference graph is not colorable. So A is selected for splitting first. A is removed from the interference graph and the flag *coalesced* is set to false for A1. Splitting A results in two cliques {A1,B,C,C1,D,D1} and {B,B1,D,D1}. The interference graph is still not colorable due to the oversized clique {A1,B,C,C1,D,D1}. Then B is then selected for splitting. The cliques after splitting B are {A1,C,C1,D,D1} and {B1,D,D1}. Note that C1 and D1 are flagged as *coalesced* and they are not counted in computing the order and colorability of the cliques. Thus, no more oversized cliques are found. So the interference graph is now colorable.

5.4 Shared Phases with Aggressive Splitting

After SplitorSpill, the same Unspill phase used in aggressive splitting is applied. The nonsplit arrays A and B are examined in this phase. Neither A nor B can be successfully unspilled. So they will not be included in the interference graph.

Finally, all the arrays that are flagged as *coalesced* can be safely discarded. The same Allocation phase (Algorithm 4) used in aggressive splitting is applied to color all arrays remaining in the interference graph. For the example program in Figure 2(a), both aggressive splitting and on-demand splitting give rise to the same SPM allocation result depicted in Figure 4.

6. Experimental results

We have implemented our two interval-coloring algorithms in the SUIF and MachSUIF compiler framework and evaluated both against our memory coloring approach [17]. For memory coloring, George and Appel's iterative-coalescing framework [10] incorporated with the generalized heuristics from [22] for handling register classes and aliases is used for coloring arrays. In all three cases, the same live range splitting heuristic discussed in this paper is applied.

For convenience, all three algorithms compared are denoted by two-letter acronyms. MC stands for memory coloring, AS for aggressive splitting and OS for on-demand splitting.

Benchmark	#Lines	#Arrays	Data Set Size (Bytes)
gsmtoast	6031	62	17.8K
gsmuntoast	6031	62	17.8K
adpcmencode	741	5	2.9K
adpcmdecode	741	5	2.9K
pegwitencode	7138	121	226.7K
pegwitdecode	7138	121	226.7K
mpeg2encode	8304	62	9.2K
mpeg2decode	9832	76	21.8K
fft00	1455	10	7.9K
autcor00	886	5	2.4K

Table 1. Benchmarks from MediaBench and EEMBC.

Table 1 gives the 10 embedded benchmarks used in our experiments, where `fft00` and `autcor00` are from EEMBC and the remaining eight programs are from MediaBench.

The profiling information is obtained using inputs that are different from those used in performance evaluations. For MediaBench, the so-called second data sets available in the MediaBench website are used for profiling while the data sets that come with their source files are used in experiments. For the two EEMBC benchmarks, `autocor00_data_3` and `fft00_data_3` are used to collect profiling information while `autocor00_data_1` and `fft00_data_1` are used for performance evaluations.

The arrays that are not accessed in a program are not considered in SPM allocation. In addition, arrays with sizes being larger than 32K bytes are ignored since it is probably ineffective to keep them entirely in SPM. One solution is to divide them into smaller subarrays [14] and then apply our algorithms to to these subarrays.

6.1 Compile Times

Table 2 gives the average compile times (calculated across all SPM sizes considered in this paper) for all three algorithms on a 2.66GHz Pentium 4 box with 2GB memory. As shown in Table 2, all three algorithms are practically efficient. For most benchmarks, AS compiles faster than OS. OS is more expensive because it starts with a larger interference graph, as is evident from Table 3.

6.2 Interference Graphs

Table 3 compares AS and OS in terms of the initial interference graphs, i.e., the ones that each starts with for the 10 benchmarks.

Benchmark	MC	AS	OS
gsmtoast	0.491	0.385	0.457
gsmuntoast	0.410	0.339	0.424
adpcmencode	0.009	0.009	0.009
adpcmdecode	0.009	0.009	0.009
pegwitencode	2.591	2.359	4.875
pegwitdecode	2.471	2.188	3.971
mpeg2encode	0.620	0.587	0.553
mpeg2decode	1.259	1.510	1.410
fft00	0.074	0.075	0.082
autcor00	0.067	0.041	0.041

Table 2. Average compile times (in seconds) under memory coloring (MC), aggressive splitting (AS) and on-demand splitting (OS).

Recall that AS starts with an interference graph including only the hot arrays in a program. On the other hand, OS starts with an interference graph including both the hot and nonsplit arrays in a program. As a result, the initial interference graph constructed by OS for a program is usually larger and contains more maximal cliques.

Benchmark	Aggressive Splitting (AS)		On-demand Splitting (OS)	
	#Maximal Cliques	Chromatic Number (Bytes)	#Maximal Cliques	Chromatic Number (Bytes)
gsmtoast	31	1672	39	2248
gsmuntoast	16	1568	19	1840
adpcmencode	1	2928	1	2936
adpcmdecode	1	2928	1	2936
pegwitencode	95	16624	267	21968
pegwitdecode	52	17056	141	26600
mpeg2encode	5	1096	15	6472
mpeg2decode	8	5768	12	11296
fft00	8	4096	10	7392
atucor00	4	308	4	2400

Table 3. Statistics on maximal cliques and chromatic numbers for the initial interference graphs constructed by AS and OS.

By comparing Column 4 of Table 1 to Columns 3 and 5 in Table 3, we find that the chromatic number, i.e., the minimal SPM size required to hold all arrays in the initial interference graph of a benchmark for either interval-coloring algorithm is much smaller than the overall data set size of the benchmark. This indicates clearly that the data set of a program can be successfully placed in an SPM with a size that is much smaller than that of the data set.

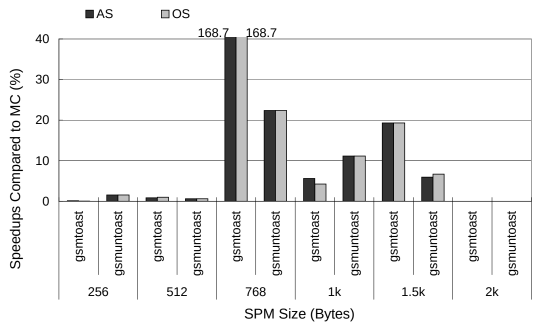
From Columns 3 and 5 in Table 3, we can also compare AS and OS in terms of the chromatic numbers of their initial interference graphs. For each benchmark, the chromatic number for AS is significantly smaller than that for OS. This suggests that the optimal solutions can be more easily found for the initial interference graphs that AS starts with. (However, this does not imply that AS will yield better performance results than OS since both start with two different interference graphs for a given program.)

6.3 Performance Evaluation

We have modified SimpleScalar to allow us to carry out performance evaluations for this work. Recall that there are four parameters involved in our cost model (Section 2.1). The cost of communicating n bytes between SPM and off-chip memory is approximated by $C_s + C_t \times n$ in cycles, where C_s is the startup cost and C_t is the cost per byte transfer. Two other parameters are M_{spm} and M_{mem}, which represent the number of cycles required for one memory access to SPM and off-chip memory, respectively. In our experiments, we have used $C_s = 100$, $C_t = 1$, $M_{mem} = 100$ and $M_{spm} = 1$.

We have evaluated the three algorithms, MC, AS and OS, for all the 10 benchmarks using a number of different SPM configurations. The allocation results from the three algorithms are the same for three benchmarks: adpcmencode, adpcmdecode and autcor00. In addition, there are only slight performance differences for mpeg2encode and mpeg2decode. For fft00, the allocation results are identical in almost every SPM configurations except when the SPM size is set to 1K bytes. In this exceptional configuration, both AS and OS have achieved a speedup of 18% over MC.

Below we present and analyze our experimental results for the remaining four benchmarks: gsmtoast, gsmuntoast, pegwitencode and pegwitdecode.



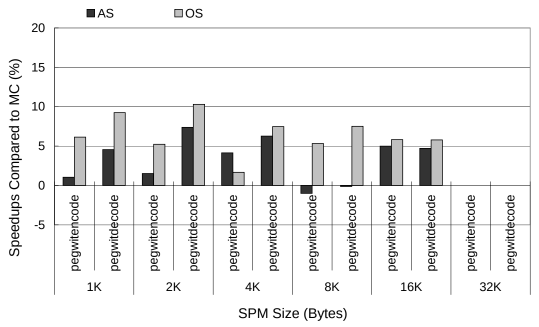
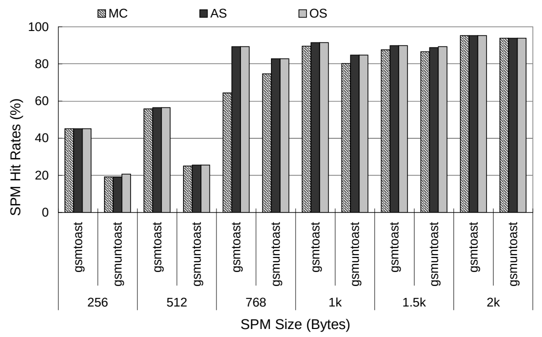


Figure 6. Speedups of aggressive splitting (AS) and on-demand splitting (OS) over memory coloring (MC).

Figures 6 shows the speedups of AS and OS over MC. The speedups for both interval-coloring algorithms are quite significant. OS outperforms MC in every SPM configuration for all four benchmarks. AS suffers a slight performance degradation in pegwitencode (1%) and pegwitdecode (0.12%) when the SPM size is 8K bytes. This is because for these two benchmarks, some frequently accessed arrays cannot be split successfully and are thus not included in the initial interference graphs constructed by AS for the two benchmarks. For both AS and OS, the largest performance improvements are observed in gsmtoast when the SPM size is set to 768 bytes. In both cases, a speedup of 168.7% has been attained. These performance advantages indicate that the colorability criterion employed in AS and OS is more accurate than that used in MC for the superperfect interference graphs considered in this work.

Let us compare OS and AS in terms of their performance results for the four benchmarks given in Figure 6. OS achieves similar results for gsmtoast and gsmuntoast as AS but outperforms AS in most of the SPM configurations used for pegwitencode and pegwitdecode. As mentioned above, some frequently accessed arrays in pegwitencode and pegwitdecode are not split. Hence, OS has delivered better results in most configurations. For pegwitencode when the SPM size is 4K bytes, OS suffers a slight performance slowdown compared to AS. This is largely due to the inaccuracy of our cost model in estimating splitting and spilling costs.

Figure 7 compares the three algorithms in terms of the SPM hit rates for the same four benchmarks given in Figure 6. For each algorithm, the hit rate for a program increases as the SPM size



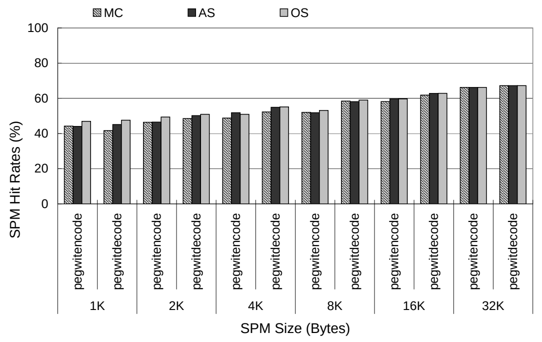


Figure 7. SPM hit rates under memory coloring (MC), aggressive splitting (AS) and on-demand splitting (OS).

increases until all the arrays in the corresponding interference graph have been placed in SPM. In almost all SPM configurations, the hit rates for OS and AS are higher than that for MC. This fact correlates well with the performance advantages of OS and AS as shown in Figure 6. When the SPM is large enough ($\geqslant$ 2K bytes for gsmtoast and gsmuntoast and $\geqslant$ 32K bytes for pegwitencode and pegwitdecode), all array accesses will hit in SPM. The SPM hit rates become identical for all three algorithms.

7. Related Work

Existing SPM allocation methods are either static or dynamic, depending on whether or not an array can be copied into and out of SPM during program execution. A large number of early methods are static. In particular, two static methods presented in [21, 2] are based on integer linear programming (ILP), which can be expensive if applied to a program with a large data set [20]

A dynamic method can often outperform an optimal static method. To our knowledge, there are four dynamic methods [15, 23, 24, 17]. In [15], loop and data transformations are exploited but the proposed technique is restricted to well-structured loop kernels. In [24], Verma et al. give an ILP formulation. In [23], Udayakumaran and Barua present a set of heuristics and apply them to a set of benchmarks. In [17], we map the SPM allocation problem into a well-understood register allocation problem.

The interval coloring problem has a fairly long history dating back, at least to 1970s [7, 8]. It has been proved that the interval coloring problem is NP-complete [9]. Fabri [7] made the connection between interval coloring and compile-time memory allocation in 1979. Since then a few approximation algorithms have been proposed [7, 16, 11], where a program is generally abstracted as a straight-line program. As a result, the interference graph for static memory objects is an interval graph [12]. With this abstraction, the interval coloring problem remains to be NP-complete [9] and the above approaches can thus provide approximate solutions. In this paper, we introduce yet another dynamic method by formulating the SPM allocation problem as an interval-coloring problem.

Interval coloring for an arbitrary graph is too complex. Recent research has focused on developing efficient interval coloring algorithms for some special graphs like chordal graphs [18, 4] and interval graphs [25]. Independently, in the field of register allocation, researchers have become increasingly more interested in abstracting interference graphs as some special graphs. For example, Andersson [1] and Pereira and Palsberg [19] have tested a large number of interference graphs and found that most interference graphs are chordal graphs. In [13], Hack et al. demonstrated that the interference graphs for programs in SSA-form [5] are chordal graphs. An optimal graph coloring is thus possible.

8. Conclusion

To our knowledge, this paper applies interval coloring to solve the SPM allocation problem for the first time. We recognize that the array interference graphs in some embedded applications can be abstracted as superperfect graphs. In a superperfect graph, its chromatic number is identical to its clique number. While the recognition of a superperfect graph is an open NP problem, we have developed two efficient and near-optimal algorithms to handle a special class of array interference graphs in which every pair of interfering arrays are containing-related. Our algorithms can achieve optimal results for such an interference graph when the size of a given SPM is no smaller than the clique number of the graph. If the SPM is not large enough, our algorithms use a new cost model to reduce the clique number of the graph by splitting or spilling some arrays from the graph until all arrays remaining in the graph can be optimally placed in SPM. Both algorithms have been implemented in the SUIF and MachSUIF compiler framework and evaluated using MediaBench and EEMBC benchmarks. Experimental results show that our interval-coloring approach can outperform our earlier memory coloring approach for some embedded applications even though memory coloring is admittedly more general and may also be effective to programs with arbitrary interference graphs.

9. Acknowledgements

This work is supported by ARC grants DP0452623 and DP0665581. The first and last authors are also supported in part by NICTA.

References

[1] Christian Andersson. Register allocation by optimal graph coloring. In *CC'03: Proceedings of the 12th International Conference on Compiler Construction*. Springer-Verlag, 2003.

[2] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[3] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES'02: Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[4] Giuseppe Confessore, Paolo Dell'Olmo, and Stefano Giordani. An approximation result for the interval coloring problem on claw-free chordal graphs. *Discrete Applied Mathematics*, 120(1-3):73–90, 2002.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL'89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[6] D. Detlefs and O. Agesen. Inlining of virtual methods. In *ECOOP'99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278, 1999.

[7] Janet Fabri. Automatic storage optimization. In *SIGPLAN'79: Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 83–91, New York, NY, USA, 1979. ACM Press.

[8] M. R. Garey and D. S. Johnson. The complexity of near-optimal graph coloring. *Journal of the ACM*, 23(1):43–49, 1976.

[9] Michael R. Garey and David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[10] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, 1996.

[11] Jordan Gergov. Algorithms for compile-time memory optimization. In *SODA'99: Proceedings of the 10th annual ACM-SIAM Symposium on Discrete algorithms*, pages 907–908, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[12] Martin Charles Golumbic. Algorithmic graph theory and perfect graphs. *Annals of Discrete Mathematics*, 2004.

[13] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In *CC'06: Proceedings of the 15th International Conference on Compiler Construction*. Springer-Verlag, 2006.

[14] Qingguang Huang, Jingling Xue, and Xavier Vera. Code tiling for improving the cache performance of PDE solvers. In *International Conference on Parallel Processing*, pages 615–625, 2003.

[15] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *DAC'01: Proceedings of the 38th Conference on Design Automation*, pages 690–695. ACM Press, 2001.

[16] H. A. Kierstead. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Mathematics*, 87(2-3):231–237, 1991.

[17] Lian Li, Lin Gao, and Jingling Xue. Memory coloring: a compiler approach for scratchpad memory management. In *PACT'05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Sriram V. Pemmaraju, Sriram Penumatcha, and Rajiv Raman. Approximating interval coloring and max-coloring in chordal graphs. *Journal of Experimental Algorithmics*, 10:2.8, 2005.

[19] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS'05: Proceedings of the 3rd Asia Symposium on Programming Languages and Systems*, pages 315–329, 2005.

[20] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, and Richard B. Brown. Compiler managed dynamic instruction placement in a low-power code cache. In *CGO'05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 179–190, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Jan Sjödin and Carl von Platen. Storage allocation for embedded processors. In *CASES'01: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 15–23. ACM Press, 2001.

[22] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI'04: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 277–288. ACM Press, 2004.

[23] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES'03: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 276–286. ACM Press, 2003.

[24] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS'04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 104–109, New York, NY, USA, 2004. ACM Press.

[25] Thomas Zeitlhofer and Bernhard Wess. List-coloring of interval graphs with application to register assignment for heterogeneous register-set architectures. *ACM Signal Processing*, 83(7):1411–1425, 2003.