Memory Coloring: A Compiler Approach for Scratchpad Memory Management

Lian Li[*], Lin Gao[†] and Jingling Xue[‡]
Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia [* † ‡]
National ICT Australia [* ‡]

Abstract

Scratchpad memory (SPM), a fast software-managed on-chip SRAM, is now widely used in modern embedded processors. Compared to hardware-managed cache, it is more efficient in performance, power and area cost, and has the added advantage of better time predictability. This paper introduces a general-purpose compiler approach, called memory coloring, to efficiently allocating the arrays in a program to an SPM. The novelty of our approach lies in partitioning an SPM into a "register file", splitting the live ranges of arrays to create potential data transfer statements between the SPM and off-chip memory, and finally, adapting an existing graph-colouring algorithm for register allocation to assign the arrays in the program into the register file. Our approach is efficient due to the practical efficiency of graph-colouring algorithms. We have implemented this work in SUIF and machSUIF. Preliminary results over benchmarks show that our approach represents a promising solution to automatic SPM management.

1 Introduction

A scratchpad memory (SPM) is a fast on-chip SRAM managed by software (the application and/or compiler). Compared to hardware-managed caches, SPMs offer a number of advantages. First, SPMs are more energy-efficient and cost-effective than caches since they do not need complex tag-decoding logic. Second, in embedded applications with regular data access patterns, an SPM can outperform a cache memory since software can better choreograph the data movements between the SPM and off-chip memory. Finally, such a software-managed data movement guarantees better timing predictability, which is critical in hard real-time embedded systems. Given these advantages, SPMs are increasingly used as an alternative to caches in modern embedded processors such as Motorola `M-core MMC221` and `TI TMS370Cx7x`. In other embedded processors such as `ARM10E` and `ColdFire MCF5`, both caches and SPMs are included in order to obtain the best of both worlds.

For SPM-based systems, the programmer or compiler must schedule explicit data transfers between the SPM and off-chip memory. The effectiveness of such an SPM management affects critically the performance and energy cost of an application. In today's industry, this task is largely accomplished manually. The programmer often spends a lot of time on partitioning data and inserting explicit data transfers required between the SPM and off-chip memory. Such a manual approach is time-consuming and error-prone. In addition, data aggregates such as arrays in large programs often exhibit cross-function data reuse. Obtaining satisfactory solutions for large applications by hand can be challenging. Finally, hand-crafted code is not portable since it is usually customised for one particular architecture.

To overcome these limitations, researchers have investigated a number of compiler strategies for allocating data to an SPM automatically. In this paper, we address the important problem of efficiently allocating arrays to an SPM, allowing arrays to be dynamically swapped into and out of the SPM during program execution. We are aware of two such dynamic methods [12, 20], where [12] is restricted to loop-oriented kernels and [20] applies to whole programs but solves the problem by resorting to integer linear programming (ILP), which may be too expensive to be useful for large real codes (considering the interprocedural nature of the problem as discussed in the preceding paragraph).

In this paper, we present a general-purpose compiler approach, called *memory coloring*, to automatically allocating the arrays in a program to an SPM. The novelty of our approach lies in partitioning an SPM into a "register file" then adapting an existing graph-coloring algorithm for register allocation to allocate the arrays in the program to the register file. We generate the data transfer statements required between the SPM and off-chip memory by splitting the live ranges of arrays based on a cost-benefit analysis. We determine whether an array should be SPM-resident or not by graph coloring. While graph-coloring has been fully studied in register allocation, this work is the first (to our knowledge) to use such a strategy for SPM management.

Our approach applies to whole programs and is scalable due to the practical efficiency of graph-colouring.

We have completed an implementation of our approach in SUIF [14] and machSUIF [17]. Preliminary results from SimpleScalar show that it represents a promising solution to the automatic SPM management problem.

The rest of this paper is organised as follows. Section 2 defines precisely the SPM management problem we address and some challenges we must overcome. Section 3 introduces our methodology for solving the problem. In Section 4, we present a concrete implementation (i.e., one instance algorithm) of our methodology in SUIF and machSUIF compilers. In Section 5, we present some preliminary results obtained from SimpleScalar over benchmark programs, demonstrating the feasibility of our methodology. Section 6 reviews the related work. In Section 7, we conclude the paper and discuss some future work.

2 Problem Statement

Given a program to be executed on an SPM-based embedded system, we address the problem of developing a compiler approach to determining the dynamic allocation and deallocation of the arrays in the program in the SPM so as to maximise the performance of the program.

The overall data set for the array candidates to be allocated to the SPM is assumed large enough so that only part of the data set can be kept in the SPM at any time during program execution. As a result, the arrays that reside in the SPM earlier may be copied back (if they are to be used later) to the off-chip memory to make room for the other arrays that will be more frequently accessed in the near future.

Therefore, there are two inter-related tasks to solve:

Task A: Mapping of Array Addresses to the SPM Space. The compiler identifies when and where an array should reside in the SPM and translates those SPM-resident arrays from their addresses in the off-chip memory to their addresses in the SPM.

Task B: Generation of Data Transfer Statements. The compiler schedules the explicit data transfers required between the SPM and off-chip memory.

The major challenge is to keep in the SPM the data that are frequently accessed in a region when that region is executed while minimising the overall data transfer cost between the SPM and off-chip memory. To this end, we need to identify the "frequently used data" at compile time since an array may have a live range spanning multiple functions and be accessed frequently only at parts of its live range.

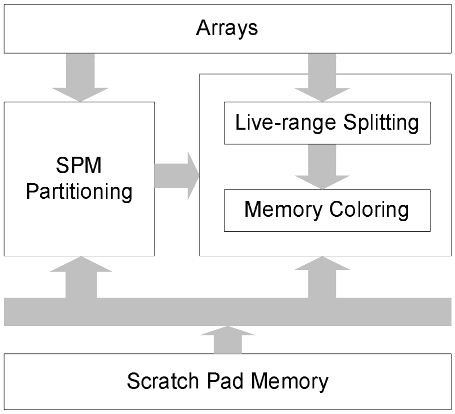


Figure 1. An SPM management methodology.

An array whose size exceeds that of the SPM can not be allocated in the SPM. Large arrays can be split into smaller "arrays" by means of loop tiling [21] and data tiling [7, 12]. Its integration with this work is part of future work.

We do not deal with scalars in this work. However, scalars can be considered as special cases of arrays. Alternatively, a scalar spill buffer can be reserved in the SPM space so that all scalar spills during register allocation for scalars can be directed to the buffer.

3 Methodology

The basic idea is to formulate the SPM management problem into one that can be solved by an existing graph-coloring algorithm for register allocation. As illustrated in Figure 1, our methodology has three main components:

SPM Partitioning. The arrays in a program considered for SPM allocation are clustered into equivalent classes, called *array classes*. All arrays in an array class are normalised to a common size. Accordingly, the SPM is partitioned (multiple times) into a register-file so that different registers thus obtained can hold arrays of different sizes. By partitioning the SPM multiple times, we introduce aliases between registers. Two registers are *aliases* if their SPM spaces overlap and *independent* otherwise. Two registers are *interchangeable* if they have the same size but disjoint SPM spaces.

Live-Range Splitting. This aims at solving Task B as stated in Section 2. An array may be frequently accessed at some parts of its live range. Based on a cost-benefit analysis, we are therefore motivated to split its live range at suitable points and insert the array copy

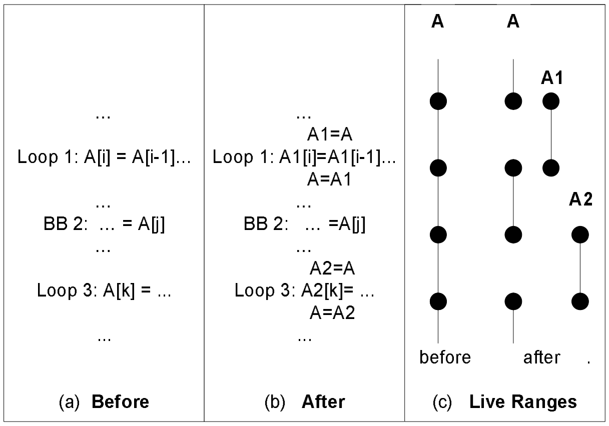


Figure 2. Live-range splitting.

statements at the splitting points. These copy statements become potentially the data transfer statements between the SPM and off-chip memory. The unnecessary copies will be eliminated by coalescing during and after graph coloring (Section 4.4). As illustrated in Figure 2, the live range of an array, A, has been split twice, possibly because the two new ranges $A1$ and $A2$ are more frequently accessed than the remaining ones. Note that the last copy statement "$A = A2$" will not be inserted if A is not live at that point.

Memory Coloring. This aims at solving Task A as stated in Section 2. The *register class* for an array class consists of all registers to which the arrays in that class can be assigned. Two register classes are *disjoint* if they do not contain a common register and *non-disjoint* otherwise. The proposed approach is flexible to embrace both disjoint and non-disjoint classes. All register classes will be mutually disjoint if all arrays in an array class of a given size are assignable only to the registers of that size. Non-disjoint register classes will result if larger registers are also permitted.

By treating the arrays (including the ones obtained after live-range splitting) as register candidates, we can adapt an existing graph-coloring algorithm such as the one in [18] to color all the arrays, resulting in each array to reside either in the SPM or the off-chip memory.

Finally, the program is modified so that the accesses to the SPM-resident arrays are accessed correctly.

A compiler-directed SPM management strategy can have difficulties in dealing with functions whose source codes are unavailable. For example, there are complications if an assembly function accesses some global arrays that happen to be allocated to the SPM by the compiler. This is because we may be unable to perform Task A as stated in Section 2 for the function. However, there will be no problems if an assembly function does not access global arrays. In embedded systems, the SPM is typically mapped into an address space that is disjoint from the off-chip memory, but connected to the same address and data buses [15]. If an array is passed from a non-assembly function to an assembly function by reference (or pointer as in C), then the address of the array (be it in the SPM or off-chip) will be passed correctly.

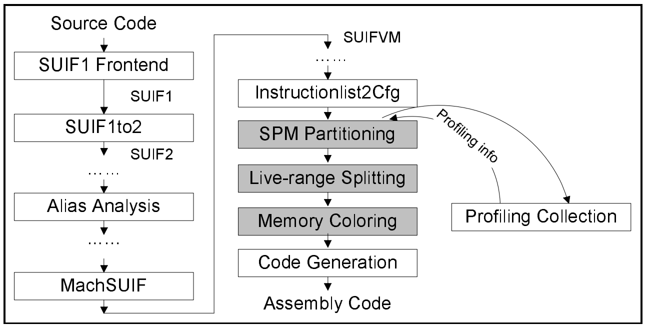


Figure 3. A concrete implementation of our methodology in SUIF and machSUIF.

4 A Concrete Implementation

Figure 3 depicts a concrete implementation of our methodology in SUIF1 and SUIF2 [14] as well as machSUIF [17]. The three components of our methodology are positioned in the boxes as highlighted in gray.

Initially, a given program is translated into an intermediate representation called the SUIF1 format using the SUIF1 frontend on the Alpha architecture. The Alpha architecture is chosen because it is supported by SimpleScalar, which will be essential for performance evaluation. The SUIF2 frontend does not support the Alpha architecture.

Once the SUIF1 format has been converted to the SUIF2 format, the SUIF2 frontend will conduct its passes including alias analysis. The alias analysis is performed based on Bjarne Stenensgaard's points-to analysis algorithm [19] implemented in SUIF2. The alias information will be used in live-range analysis and live-range splitting.

Next, the SUIF2 format is converted to the SUIFVM format using machSUIF, a backend developed for the SUIF compilers [17]. The SUIFVM format for a function is then translated into the CFG (control flow graph) for that function. Based on the profiling infrastructure provided by machSUIF, we have added a profiling module to gather the frequencies in which all arrays are accessed in a program.

Our method operates on the CFGs of the functions in a

```
1  #DEFINE ALIGN_UNIT = a tunable constant (in bytes)
2  PROCEDURE SPM_Partitioning()
3  // Part I: define array classes
4  Let $\mathcal{A}$ be the set of all arrays that are to be allocated to SPM
5  for every array $A$ in $\mathcal{A}$
6      Let $A.size$ be its (declared) size (in bytes)
7      $A.aligned\_size = \lceil \frac{A.size}{\text{ALIGN\_UNIT}} \rceil * \text{ALIGN\_UNIT}$
8  Define $ArrayClass_n = \{A \in \mathcal{A} \mid A.aligned\_size = n\}$
9  Set $ArrayClass_n.size = n$
10 Define $ArrayClassSet = \cup_{n \in \{A.aligned\_size | A \in \mathcal{A}\}} \{ArrayClass_n\}$
11 // Part II: define the register file
12 for every array class $A_c$ in $ArrayClassSet$
13     for ($i = 0$; $i < \lfloor \frac{SPM\_SIZE}{A_c.size} \rfloor$; $i$ ++)
14         $start\_addr = SPM\_BASE + i * A_c.size$
15         Create register $R_{A_c.size,ID}$, where $ID = i$
16         Set $R_{A_c.size,ID}.addr = start\_addr$
17         Set $R_{A_c.size,ID}.size = A_c.size$
18         $RegClass_{A_c.size} \cup = \{R_{A_c.size,ID}\}$
19     $PRF \cup = \{RegClass_{A_c.size}\}$
```

Figure 4. An algorithm for SPM partitioning.

program. We first give an overview of our implementation and then describe the three components of our method.

4.1 An Overview

In our current implementation, we consider programs free of recursion. No previous method can place any data in recursive functions either. Recursive functions can be handled if we adopt the caller-callee save register mechanism used for scalars also for arrays. Since there are no recursive functions, we will treat local and global array objects identically during graph coloring. As we shall see in Section 4.4, this will affect how the live ranges of arrays are defined.

The alias information is used in live-range analysis and splitting. Aliases will not affect the address translations performed in Task A as stated in Section 2. In programs such as those written in C, pointers create aliases with arrays. A pointer p to an array A is always initialised in the form of $p = A + offset$ (in C). So making the array A SPM-resident causes p to point the SPM-resident array correctly.

4.2 SPM Partitioning

Figure 4 gives a simple algorithm for partitioning an SPM of size, *SPM_SIZE*, into a pseudo register file, denoted by the set *PRF* (in bytes). Let *SPM_BASE* be the start address of the SPM space (line 14). This algorithm has two parts. In Part I (lines 3 – 10), we cluster all the arrays in the program into *array classes* such that the arrays in the same class have the same aligned (or normalised) size. The motivation for using a tunable parameter (line 1), ALIGN_UNIT, is to avoid introducing a large number of array classes containing arrays with similar sizes, resulting in an unnecessarily large register file. On the other hand, the larger ALIGN_UNIT is, the worse the SPM space will be utilised. In Part II (lines 11 – 19), we divide the SPM space (multiple times) by creating the (pseudo) registers for holding arrays, the register classes for array classes and a register file for the SPM. For every array class A_c, the SPM is sliced into $\lfloor \frac{SPM\_SIZE}{A_c.size} \rfloor$ consecutive chunks starting from its beginning. These chunks are the so-called *(pseudo) registers* to which the arrays in A_c can be assigned. The register class for A_c consists of $\lfloor \frac{SPM\_SIZE}{A_c.size} \rfloor$ such registers. The remaining $SPM\_SIZE - \lfloor \frac{SPM\_SIZE}{A_c.size} \rfloor A_c.size$ bytes in the SPM are unused by this register class. There are as many register classes as there are the number of array classes: $|PRF| = |ArrayClassSet|$ (lines 18 – 19). The register file, *PRF*, is the set of these register classes.

According to this SPM partitioning algorithm, two registers in the same register class are never aliases and all register classes are mutually disjoint. However, the registers in different register classes can be aliases.

Figure 5 illustrates our algorithm using an example. We start with seven arrays in the program shown in Figure 5(a) and end up with five array classes as shown in Figure 5(b). Figure 5(c) depicts the five register classes obtained (with the SPM space divided five times). For example, $R_{320,1}$ and $R_{256,1}$ are aliases since their SPM spaces overlap.

4.3 Live-Range Splitting

In order to keep frequently accessed arrays in the SPM, we adopt the idea of live-range splitting used for scalars in the recent register allocation work [2] for arrays. The objective is to solve Task B as stated in Section 2 and illustrated in Figure 2. By splitting the live ranges of some selected arrays, we introduce copy statements that will become potentially data transfer statements between the SPM and off-chip memory. As we shall see in Section 4.4, we will eliminate unnecessary array copies during and after graph coloring.

In this initial study, we focus on the arrays that are frequently accessed in loops. The basic idea is to split the live range of a frequently accessed array in a loop nest. The array is copied to a new array at the earlier splitting point (at the beginning of the loop) and restored back at the later splitting point (at the end of the loop). During memory coloring, all these new arrays will be candidates to be colored first so that they will likely be allocated to the SPM.

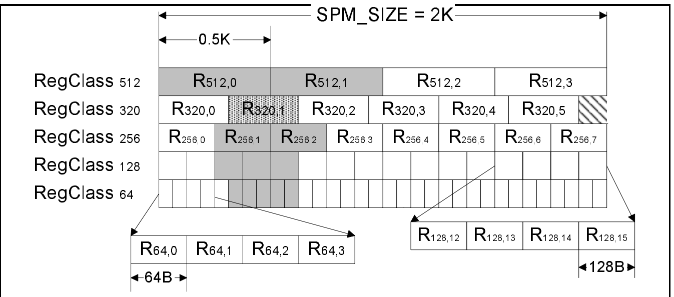
We use a cost-benefit analysis to identify the arrays whose live ranges can be split beneficially. Our cost model takes into account the access frequencies of arrays (obtained by runtime profiling) and the data transfer cost between the SPM and off-chip memory. The cost of communicating n bytes between the SPM and off-chip memory is typically approximated by $C_s + C_t \times n$ [12] (cycles), where C_s is the

Array	Size (Bytes)
A	512
B	320
C	256
D	104
E	100
F	18
G	8

(a) Arrays

Array Class	Size (Bytes)
$\{A\}$	512
$\{B\}$	320
$\{C\}$	256
$\{D, E\}$	128
$\{F, G\}$	64

(b) Array classes



(c) Register file

Figure 5. An illustration of the SPM partitioning algorithm in Figure 4 ($SPM\_BASE = 0$, $SPM\_SIZE = 2KB$ and ALIGN_UNIT $= 64B$). The last portion in the SPM space is unused by $RegClass_{320}$.

startup cost and C_t the transfer cost per byte. We write S_{spm} and M_{mem} for the number of cycles required per array element access to the SPM and off-chip memory, respectively.

Figure 6 gives an algorithm, *Live_Range_Splitting*, that operates on the CFG of a function. To simplify the presentation of this algorithm, every call site is assumed to be contained in a loop nest (since it could be made so trivially otherwise). In line 2, we process all the loop nests in a function one by one. In line 3, we examine all the loops of a particular loop nest, starting from its outermost to innermost loop. We will split the live range of an array A with respect to a loop L (line 4) at most once (line 5). We skip A if *CanSplit*(A, L) returns false (line 6) since it can be generally difficult to perform the code rewriting in lines 30 and 33. In line 7, we check if it is beneficial to split the live range of A. In the function *SplitCost*, *num_of_copies* is set to 1 or 2 depending on the dynamic number of copy statements executed (lines 29 and 32). If the splitting is beneficial, *Split_and_Copy* is called in line 8 to split the live range of A. In line 28, a new array A' is introduced, and it is made to inherit the same SPM partitioning information from A. In lines 29 and 32, the copy statement(s) required are inserted as indicated. In line 30, all the accesses to A (explicit or implicit (via pointers pointing only to A) inside L are changed to the accesses to A'. In line 33, any pointer that pointed to A (uniquely due to lines 10 – 11) is restored to point to A again if it is visible outside the loop L.

Figure 7 illustrates our live-range splitting algorithm using a double loop taken from a Media benchmark program. Figure 7(a) shows the double loop. In the CFG for the double loop given in In Figure 7(b), the outer loop is shown but the basic blocks for the inner loop are suppressed. Figure 7(c) shows that the live ranges of two arrays, `rpf` and `uf` are split. The corresponding new arrays, `rpf_1` and `uf_1`, are introduced. The copy statements required are inserted in the basic blocks `BBi` and `BBj` as shown. (According to the algorithm, the copy statement inserted for `uf` in `BBj` should have been inserted on its incoming edge. No copy statement is needed for `rpf` since it is not modified.)

4.4 Memory Coloring

Given the register file and array candidates as defined in *SPM_Partitioning* (including also the new arrays introduced by *Live_Range_Splitting*), we determine which arrays should reside in which parts of the SPM by adapting an existing graph-coloring algorithm for scalars. This solves Task A as stated in Section 2. Recall that the live-range splitting we discussed earlier aims at solving Task B.

Section 4.4.1 describes our live-range analysis for arrays (local or otherwise), which is interprocedural and needs to be carried out only once for a program. Section 4.4.2 gives our memory coloring algorithm for arrays.

4.4.1 Live-Range Analysis

The live ranges of all arrays are required in order to construct the interference graphs used during memory coloring. Due to the global nature of memory coloring, we extend the live-range analysis for scalars to compute the live ranges of arrays interprocedurally. The predicates, DEF and USED, local to a basic block $\mathcal{B}$ for a particular array A are:

$\text{DEF}_A(\mathcal{B})$ returns true if A is killed in block $\mathcal{B}$ by a copy
statement introduced in *Split_and_Copy*
$\text{USED}_A(\mathcal{B})$ returns true if the elements of A are read
or written (possibly via pointers) in block $\mathcal{B}$

By convention, the CFG of a function is assumed to have a unique entry block, denoted *ENTRY*, and a unique exit block, denoted *EXIT*. These are pseudo blocks that do not contains instructions. The standard data-flow equations that are applied to an array A in a function are given by:

$$\text{LIVEIN}_A(\mathcal{B}) = (\text{LIVEOUT}_A(\mathcal{B}) - \text{DEF}_A(\mathcal{B})) \vee \text{USED}_A(\mathcal{B})$$
$$\text{LIVEOUT}_A(\mathcal{B}) = \bigvee_{\mathcal{S} \in \text{succ}(\mathcal{B})} \text{LIVEIN}_A(\mathcal{S}) \qquad (1)$$

```
1  PROCEDURE Live_Range_Splitting()
2  for every $N$-dimensional loop nest in a function
3      for every loop $L$ starting from outermost to innermost
4          for every array $A$ accessed inside $L$
5              if $A$ has been already split in this nest continue
6              if $\neg$CanSplit$(A, L)$ continue
7              if SplitCost$(A, L) \geq$ SplitBenefit$(A, L)$ continue
8              Split_and_Copy$(A, L)$
9  BOOLEAN FUNCTION  CanSplit$(A, L)$
10  if there exists a pointer access in the loop body of $L$ such that
        the pointer may point to both $A$ and a distinct array $B$
11      return false
12  if $A$ is a global array and accessed in a function that may be
        called inside $L$ directly or indirectly
13      return false
14 return true
15 INT FUNCTION SplitCost$(A, L)$
16 SplitFreq = frequency of the pre-header of $L$
17 if $A$ is modified in $L$
18     num_of_copies = $2 \times$ SplitFreq
19 else
20     num_of_copies = SplitFreq
21 SplitCost = $(C_s + C_t \times A.aligned\_size) *$ num_of_copies
22 return SplitCost
23 FUNCTION SplitBenefit$(A, L)$
24 AccessFreq = access frequency of $A$ in $L$
25 SplitBenefit = AccessFreq $\times$ $(M_{mem} - M_{spm})$
26 return SplitBenefit
27 PROCEDURE Split_and_Copy$(A, L)$
28 Create a new array $A'$ and insert $A'$ into the same array class
   as $A$ (Figure 4), where $A'.aligned\_size = A.aligned\_size$
29 Add $A' = A$ (for array copy) at the pre-header of $L$
30 Replace every access of $A$ by an access to $A'$ in $L$
31 Add the following code on the outgoing edges of all $L$'s exits:
32     $A$=$A'$ // when $A$ is modified in $L$
33     Code for restoring a pointer to $A$ // when modified & visible
```

Figure 6. An algorithm for live-range splitting.

where succ$(\mathcal{B})$ denotes the set of all successor blocks of $\mathcal{B}$. By convention, LIVEIN$_A$ is initialised to false for all blocks.

To permit the data reuse information to be propagated across the functions, two additional sets of equations are introduced next. For convenience, we assume that each call statement forms a basic block by itself. Let *CallSite* be the set of all call statement blocks in a program. Let $F_{\mathcal{B}}$ be the set of functions invoked at the call statement block $\mathcal{B}$.

An array A is live on entry to a call statement block if it is live on entry to a callee function invoked from the call site (note that A could be accessed via pointers in the callee):

$$\begin{aligned}&\forall\, \mathcal{B} \in \mathit{CallSite} : \forall\, f \in F_{\mathcal{B}} : \\ &\mathsf{LIVEIN}_A(\mathcal{B}) \vee= \mathsf{LIVEIN}_A(f's\ \mathit{ENTRY})\end{aligned} \tag{2}$$

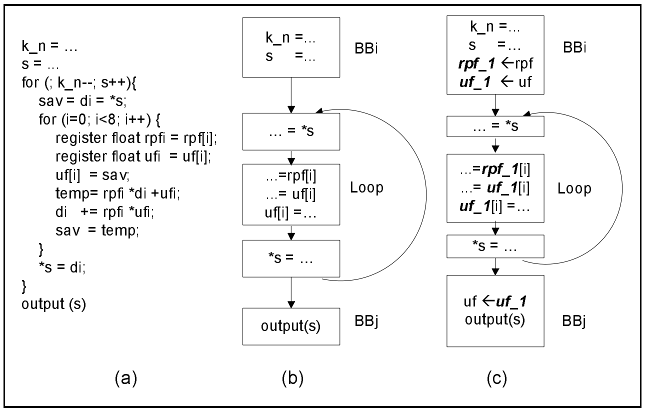

```
k_n = ...
s = ...
for (; k_n--; s++){
    sav = di = *s;
    for (i=0; i<8; i++) {
        register float rpfi = rpf[i];
        register float ufi  = uf[i];
        uf[i]  = sav;
        temp= rpfi *di +ufi;
        di   += rpfi *ufi;
        sav  = temp;
    }
    *s = di;
}
output (s)
```

Figure 7. Live-range splitting in a program.

Presently, we do not use caller-callee register saving. an array A that is live out of a call site is assumed to be live on entry of the exit of every function invoked at the call site:

$$\begin{aligned}&\forall\, \mathcal{B} \in \mathit{CallSite} : \forall\, f \in F_{\mathcal{B}} : \\ &\mathsf{LIVEIN}_A(f's\ \mathit{EXIT}) \vee= \mathsf{LIVEOUT}_A(\mathcal{B})\end{aligned} \tag{3}$$

4.4.2 Algorithm

Our algorithm, *Memory_Coloring*, given in Figure 8 is an adaptation of a generalised graph-coloring algorithm for irregular register architectures [18], which is implemented in machSUIF [17] on top of an iterated-coalescing framework described in [1]. Therefore, *Iterative_Coalescing* invoked in our algorithm is essentially the procedure "Main" described in [1, p. 251] and will thus not be discussed in this paper.

Standard graph-coloring algorithms process functions separately since they rely on caller-callee register saving to handle the live ranges across call sites. Presently, such a mechanism is not used. Instead, our algorithm operates on the call graph of a program interprocedurally. As a result, we only need to compute (interprocedurally) the liveness information for a program once. Thus, the procedure "LivenessAnalysis" invoked in "Main" [1, p. 251] is not needed.

Our algorithm performs two graph-coloring passes on a program. This is realised by calling *ColorProgram* twice with different array candidate sets. In lines 2 – 3, A.*reg* for every array A is initialised to -1 to indicate that A has not been colored, i.e., register-allocated. *ArrayClassSet* is the set of array classes defined in *SPM_Partitioning* and later extended in line 28 of *Live_Range_Spitting*. Here, we have abused the notation by writing *ArrayClassSet* to mean the set of all arrays extracted from *ArrayClassSet*.

In the first call to *ColorProgram* (line 5), only the new arrays obtained due to live-range splitting are considered. These are frequently accessed arrays. Thus, we try to allocate them to the SPM space first. In the second pass, *Color-*

```
1  PROCEDURE Memory_Coloring()
2  for every array $A$ in ArrayClassSet in the program
3      $A.reg = -1$
4  Let HotArrays be the set of all new arrays that are
   introduced due to live-range splitting in Figure 6
5  ColorProgram(HotArrays)
6  ColorProgram(ArrayClassSet)
7  Rewrite_Program()
8  PROCEDURE Color_Program(ArrayCandidates)
9  for every function $f$ in the call graph of the program
10     Iterative_Coalesce()
11     CoalesceSpill()
12     for every $A \in$ ArrayCandidates that has been colored
13         $A.reg$ = the color assigned
14 PROCEDURE Build()
15 Build the interference graph such that $A \in$ ArrayCandidates
   is in the graph if (1) $A$ is live in the function $f$ or (2) $A$ and
   $B$ interfere, where $B \in$ ArrayCandidates is in the graph
16 PROCEDURE Iterative_Coalesce()
17 Build()
18 MakeWorklist()
19 do
20     if simplifyWorklist $\neq \emptyset$ Simplify()
21     if moveWorklist $\neq \emptyset$ Coalesce()
22     if freezeWorklist $\neq \emptyset$ Freeze()
23     if spillWorklist $\neq \emptyset$ SelectSpill()
24 while simplifyWorklist $= \emptyset \wedge$ moveWorklist $= \emptyset \wedge$
         freezeWorklist $= \emptyset \wedge$ spillWorklist $= \emptyset$
25 AssignColors()
26 if SpilledNodes$\neq \emptyset$
27     for every array $A$ in SpilledNodes
28         $A.spill = true$
29         Iterative_Coalesce()
30 PROCEDURE Rewrite_Program()
31 for every array $A$ in ArrayClassSet
32     if $A.reg \neq -1$
33         Replace all occurrences of $A$'s base address by
           the base address of the register $A.reg$
```

Figure 8. An algorithm for memory coloring.

Program is called again with all array candidates extracted from *ArrayClassSet*. During this second call (line 6), *ColorProgram* will attempt to allocate to the SPM the arrays that are not yet colored in the first call.

ColorProgram processes all functions in the call graph of a program one by one (line 9). In line 10, *Iterative_Coalesce* (from [1]) is called to perform graph coloring for all array candidates in *ArrayCandidates* with respect to f. In line 17, *Build* constructs the interference graph for f (line 15). Note that *Iterative_Coalesce* is also called recursively in line 29. Thus, all those arrays whose spill flags are true are ignored during all subsequent invocations of *Iterative_Coalesce*. An array A that is *pre-colored* (when $A.reg \neq -1$) is dealt with in the standard manner.

Two arrays are *move-related* if one is obtained as a result of splitting the live range of the other. The corresponding copy statement(s) introduced by *Live_Range_Splitting* will be eliminated when the two move-related arrays are coalesced during graph coloring (line 21).

Benchmark	Data Size (Bytes)	#Arrays	#Lines
rawcaudio	2.9K	5	1019
rawdaudio	2.9K	5	1019
g721decode	1.1K	26	1704
g721encode	1.1K	26	1704
toast	17.8K	62	6031
untoast	17.8K	62	6031
queens	2.5K	5	850
bj	13K	19	2109

Table 1. Application programs.

When *AssignColors* is called in line 25, we will select the color that has the smallest number of register aliases and pick one of such registers with the smallest ID when is a tie. This tends to improve the colorability of the other arrays.

If an array is "spilled" (line 26), we simply set its spill flag to true (line 28), indicating that the array will be ignored when *Iterative_Coalesce* is called recursively next time (line 29). There is no need to generate any spill code. By removing a node from the interference graph, more coalescing opportunities may be created. Thus, the recursive calls made in line 29 can help eliminate more unnecessary array copies that may be introduced due to live-range splitting.

After *Iterative_Coalesce* returns to the invocation site in line 10, we call *CoalesceSpill* in line 11 to coalesce all the "spilled" arrays. Essentially, this undoes the effect of live-range splitting by removing the associated copy statements inserted before. In lines 12 – 13, we update $A.reg$ for every colored array so that the information will be used when *Iterative_Coalesce* is called to process the next function.

5 Experimental Results

We evaluate this work using the eight benchmarks given in Table 1. The first six are from the Media benchmark suite, `queens` is a program for solving the N-queens problem and `bj` is a program for the BlackJack game. The data size of a benchmark accounts for the space taken by only the arrays in the application of the benchmark.

All programs are compiled into assembly programs for the Alpha architecture using our implementation depicted in Figure 3. These assembly programs are then translated into binaries on a DEC Alpha 20264 architecture. The profiling information for the Media benchmarks is obtained using the so-called "second data sets" available in the Mediabench web site. These benchmarks are evaluated using the data sets that come with their source files. The profiling for the other two benchmarks is obtained using inputs different

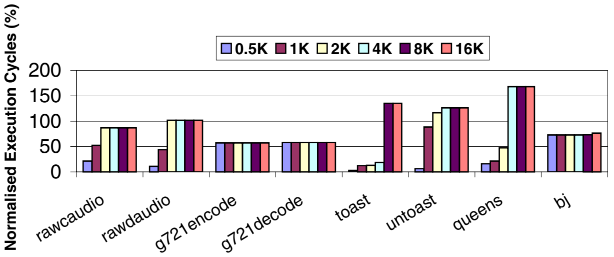


Figure 9. Effect of varying the size of the SPM on runtime gain.

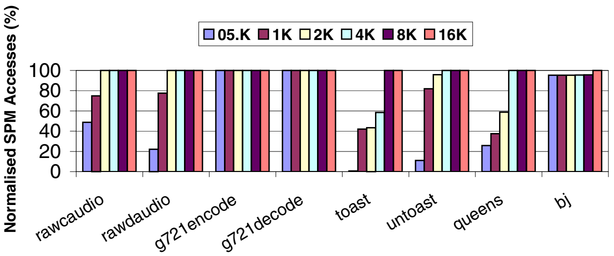


Figure 10. Effect of varying the size of the SPM on the SPM accesses.

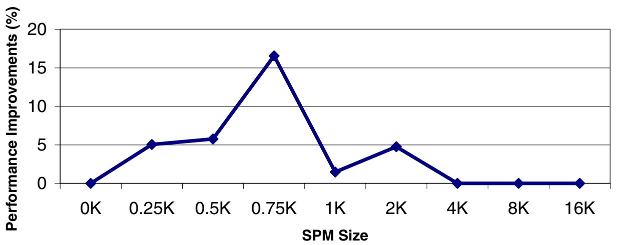


Figure 11. Effect of splitting on runtime gain.

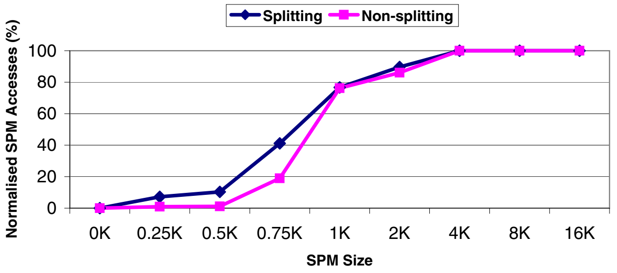


Figure 12. Effect of splitting on the SPM accesses.

from those when they are actually evaluated.

We have modified SimpleScalar to allow us to carry out performance evaluations for this work. Recall that there are four parameters involved in an SPM-based embedded system. The cost of communicating n bytes between the SPM and off-chip 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 the SPM and the off-chip memory, respectively. The values of the four parameters are $C_s = 20$, $C_t = 1$, $M_{mem} = 20$ and $M_{spm} = 1$ unless otherwise specified.

In rawcaudio and rawdaudio, there is a single loop iterating over an array of 2K bytes. We have manually tiled the loop so that the array is split into four equally sized arrays of 512B each. This creates the arrays of data sizes compatible with the other benchmarks so that they can be evaluated using some common sizes for the SPM. Unless otherwise specified, by rawcaudio and rawdaudio, we mean the tiled versions obtained this way.

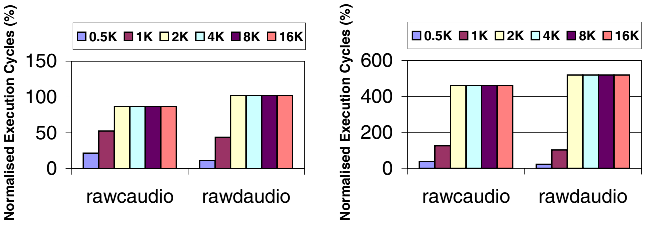
5.1 Performance Improvements

Figure 9 illustrates the performance improvements of the eight benchmarks as the size of the SPM, *SPM_SIZE*, increases. The execution time of a benchmark is normalised to that achieved when the SPM is not used. As *SPM_SIZE* increases, all eight benchmarks show non-decreasing performance improvements. Each of the eight benchmarks arrives at the best speedup at one of the SPM sizes used.

However, for some benchmarks such as g721decode and g721encode, once *SPM_SIZE* has reached a certain value, no further performance improvements are observed even when their data size, 1.1KB, as shown in Table 1 are still larger than some SPM sizes (e.g., 0.5KB and 1KB). The reasons behind can be explained using the SPM accesses as shown in Figure 10 normalised to that achieved in the ideal setting when all the array accesses (from the array candidates considered) are made to the SPM. When $SPM\_SIZE \geqslant 0.5KB$, all array accesses to the SPM are already maximised . Any further increase in *SPM_SIZE* will not have any impact on performance improvement. Finally, we observe from Figure 10 for each benchmark, all array accesses (from the array candidates considered) are eventually made to the SPM for a certain SPM size.

5.2 Impact of Live-Range Splitting

In this work, live-range splitting aims at improving graph colorability, thereby increasing the number of arrays allocated to the SPM space. Figure 11 evaluates the impact of live-range splitting on the runtime gains for untoast. When $SPM\_SIZE \geqslant$ 4KB, all the array candidates can be allocated to the SPM without resorting to live-range splitting. So live-range splitting needs not to be performed in these cases. However, we observe from Figure 11 that split-



(a) $C_s = 20$ and $M_{mem} = 20$ (b) $C_s = 30$ and $M_{mem} = 100$

Figure 13. Effect of varying C_s and M_{mem} on runtime gain ($C_t = M_{spm} = 1$).

ting is beneficial when the SPM is smaller. The resulting performance improvements are attributed to the increased SPM accesses as shown in Figure 12. For this particular benchmark, the number of live ranges split is 16. The largest interference graph consists of 33 nodes.

While the coalescing heuristics used in the iterative-coalescing algorithm [10] are designed to reduce unnecessary register-move instructions, there is no guarantee that all will be eliminated. These move instructions will be translated into array copy operations within the SPM. For example, when $SPM_SIZE = 2$KB, one live range suffers from this problem. We plan to develop coalescing heuristics that are well suited to data aggregate management.

5.3 Impact of Architecture Parameters

Figure 13 illustrates the impact of varying startup cost (C_s) and DRAM latency (M_{mem}) on the runtime gains of the two benchmarks, `rawcaudio` and `rawdaudio`. The execution times are all normalised to that achieved in the worst setting when the SPM is not used. This experiment demonstrates that our current memory coloring algorithm is capable of taking into account the architectural parameters into account when allocating arrays to the SPM. In both configurations, our algorithm finds the optimal solution as soon as SPM_SIZE increases to 2KB or beyond. Better performance speedups are attained as M_{mem} increases.

5.4 Impact of Loop and Data Tiling

We evaluate the impact of loop and data tiling on runtime improvements. In this experiment, `untiled_rawcaudio` is the original program while `rawcaudio` is the tiled program obtained as we discussed above. Figure 14 compares the execution times of both programs when *SPM_SIZE* varies. The tiled program performs better than the untiled version when SPM_SIZE is below 4KB. As soon as SPM_SIZE reaches 4KB, tiling enjoys no benefit since all the arrays can now be kept in the SPM even when the program is not tiled. In fact, the performance of the tiled program worsens slightly due to the tiling overhead introduced.

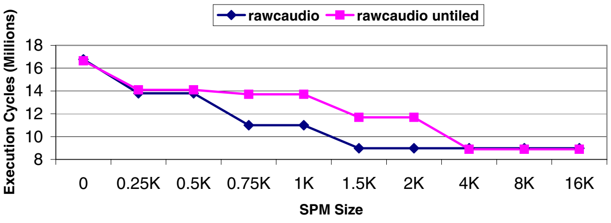


Figure 14. Effect of tiling on execution time.

This experiment suggests that loop and data transformations such as tiling should be integrated into our memory coloring framework in future work.

6 Related Work

There are a number of research efforts on allocating program data among different non-cached memory banks [3, 11, 12, 16, 20]. Most of these existing methods are static in the sense that an array will reside either in the SPM or SDRAM throughout the program execution. To our knowledge, there are two dynamic methods [12, 20], by which an array may be copied into and out of the SPM during program execution. In [12], loop and data transformations are exploited but the proposed technique is restricted to well-structured kernels. In [20], the SPM management problem is formulated as an integer linear programming (ILP) problem and the proposed approach is evaluated using small programs. ILP can be expensive if applied to large programs with arrays whose live ranges span multiple functions. Its feasibility for larger programs remains to be demonstrated.

Graph-coloring is a popular technique in register allocation. Based on Chaintin's original formulation [6], a variety of graph-coloring-based register allocators have been developed [5, 8, 10, 13, 18]. In particular, George and Appel [10] introduce a well-known iterative-coalescing algorithm. Recently, Smith, Ramsey and Holloway [18] present a generalised algorithm for irregular architectures with register aliases and non-disjoint register classes, which we have adapted to allocate arrays to an SPM in this work.

An important advance in the field of graph-coloring-based register allocation is that the live ranges of variables should be split into small pieces, with copy instructions connecting the pieces [2, 4]. A register allocator is responsible for eliminating the redundant copies introduced due to live-range splitting. We have adopted this idea in this work.

Cooper and Harvey [9] describe a technique that targets spilled scalars into a small region of an SPM. This can be done together with our technique for allocating arrays.

7 Conclusion

In this paper, we have presented a new methodology for automatically allocating arrays in a program to an SPM. We transform the SPM management problem into one that can be solved efficiently by existing graph-coloring algorithms. The basic idea is to partition the SPM space into a register file with registers capable of holding arrays of different sizes in the program. This leads to an efficient solution to the SPM management problem by divide and conquer. By splitting the live ranges of frequently accessed arrays, we introduce copy statements that represent potentially data transfer statements between the SPM and off-chip memory. The number of unnecessary splits is reduced by copy coalescing during and after graph coloring. This solves Task B stated in Section 2. By adapting existing graph-coloring algorithms, we are able to determine efficiently which arrays should be SPM-resident and where. This solves Task A stated in Section 2.

We have presented one implementation of our methodology in SUIF and machSUIF. Preliminary results from benchmarks are very encouraging. The strategies for SPM partitioning and live-range splitting as discussed in this paper are simple with a lot of room for improvement. Despite this, the prototyping implementation shows that our methodology is capable of producing optimal performance results for the benchmarks used.

There are a number of interesting but challenging research directions, including better strategies for SPM partitioning, live-range splitting and memory coloring. For example, more sophisticated heuristics for live-range splitting discussed in [2] may be considered in future work. Better coalescing heuristics are needed to minimise the number of unnecessary splits for arrays. We will also investigate how to combine loop and data transformations (e.g., tiling) in our framework for more effective SPM management. Allocation of heap data, together with arrays, to the SPM space will be yet another challenging topic.

References

[1] A. J. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[2] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 243–253, New York, NY, USA, 2001. ACM Press.

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.

[4] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[5] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[6] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, New York, NY, USA, 1982. ACM Press.

[7] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layout for hierarchical memory systems. In *ACM International Conference on Supercomputing (ICS'99)*, pages 444–453, Rhodes, Greece, Jun. 1999.

[8] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[9] K. D. Cooper and T. J. Harvey. Compiler-controlled memory. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1998. ACM Press.

[10] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[11] J. D. Hiser and J. W. Davidson. Embarc: an efficient memory bank assignment algorithm for retargetable compilers. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 182–191. ACM Press, 2004.

[12] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th conference on Design automation*, pages 690–695. ACM Press, 2001.

[13] G.-Y. Lueh, T. Gross, and A.-R. Adl-Tabatabai. Fusion-based register allocation. *ACM Trans. Program. Lang. Syst.*, 22(3):431–470, 2000.

[14] The SUIF Compiler Group. SUIF: An infrastructure for research on parallelizing and optimizing compilers. http://suif.stanford.edu.

[15] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European conference on Design and Test*, page 7. IEEE Computer Society, 1997.

[16] J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 15–23. ACM Press, 2001.

[17] M. Smith. Extending SUIF for machine-dependent optimizations, 1996.

[18] M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288. ACM Press, 2004.

[19] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM Press, 1996.

[20] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286. ACM Press, 2003.

[21] M. J. Wolfe. Iteration space tiling for memory hierarchies. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia PA, 1987.