Compiler-Directed Scratchpad Memory Management via Graph Coloring

Lian Li, Hui Feng and Jingling Xue
University of New South Wales

Scratchpad memory (SPM), a fast on-chip SRAM managed by software, is widely used in embedded systems. This paper introduces a general-purpose compiler approach, called *memory coloring*, to assign static data aggregates such as arrays and structs in a program to an SPM. The novelty of this approach lies in partitioning the SPM into a pseudo register file (with interchangeable and aliased registers), splitting the live ranges of data aggregates to create potential data transfer statements between SPM and off-chip memory, and finally, adapting an existing graph coloring algorithm for register allocation to assign the data aggregates to the pseudo register file. Our experimental results using a set of 10 C benchmarks from MediaBench and MiBench show that our methodology is capable of managing SPMs efficiently and effectively for large embedded applications. In addition, our SPM allocator can obtain close to optimal solutions when evaluated and compared against an existing heuristics-based SPM allocator and an ILP-based SPM allocator.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; B.3.2 [**Memory Structures**]: Design Styles—*Primary memory*; C.3 [**Special-Purpose And Application-Based Systems**]: Real Time and Embedded Systems

General Terms: Algorithms, Languages, Experimentation, Performance

Additional Key Words and Phrases: Scratchpad memory, software-managed cache, memory allocation, graph coloring, memory coloring, live range splitting, register coalescing

1. INTRODUCTION

Scratchpad memory (SPM) is a fast on-chip SRAM managed by software (the application and/or compiler). Compared to hardware-managed cache, it offers a number of advantages. First, SPMs are more energy-efficient and cost-efficient 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 SPM and off-chip memory. Finally, such a software-managed strategy guarantees better timing predictability, which is critical in hard real-time systems. Given these advantages, SPMs have been increasingly incorporated in modern embedded systems. In some embedded processors such as Motorola Dragonball, Infineon XC166 and TI TMS370CX7X, SPMs are used as an alternative to caches. In other embedded processors like ARM10E and ColdFire MCF5, both caches and SPMs are included in order to obtain the best of both worlds. In this work, SPMs should be seen in a general context. For example, in stream processors such as Imagine [Kapasi

Authors' address: Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia. The first and last authors are also affiliated with National ICT Australia (NICTA).

et al. 2002], a software-managed stream register file (SRF) is used to stage data to and from off-chip memory. In IBM's Cell architecture, each co-processor (known as SPE) has a software-managed local store for keeping both instructions and data.

For SPM-based systems, the programmer or compiler must schedule explicit data transfers between 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 programmers often spend a lot of time on partitioning data and inserting explicit data transfers required between SPM and off-chip memory. Such a manual approach is often time-consuming and error-prone. Moreover, the hand-crafted code is not portable since it is usually customized to a particular architecture.

To overcome these limitations, we propose a general-purpose compiler approach, called *memory coloring*, to determining the dynamic allocation and deallocation of static data aggregates such as global and stack-allocated arrays and structs in a C-like program so as to maximize the performance of the program. Whenever we speak of arrays in this paper, we mean both kinds of data aggregates. An array whose size exceeds that of the SPM under consideration cannot be placed entirely in the SPM. Such arrays can be tiled into smaller "arrays" by means of loop tiling [Xue 2000; Wolfe 1989] and data tiling [Kandemir et al. 2001]. In the proposed approach, the continuous space of an SPM is partitioned into a pseudo register file (with interchangeable and aliased registers). The data aggregates are the register candidates to be assigned to the SPM via a generalized graph coloring allocator, which can be any traditional graph coloring allocator generalized as described in [Smith et al. 2004] to handle interchangeable and aliased registers. Unlike scalars, data aggregates typically have longer live ranges. So live range splitting may be used beneficially to split their live ranges into smaller pieces. The splitting points are the places to insert all required data transfers between SPM and off-chip memory. During the coloring phase, register coalescing is applied to reduce unnecessary data transfers that would otherwise have been introduced into the final program.

In summary, this paper makes the following contributions. First, we introduce a *memory coloring* methodology, by which the SPM management problem is mapped into the well-known classic register allocation problem. This includes a scheme to partition an SPM into a pseudo register file, an interprocedural liveness analysis for data aggregates, and an algorithm to split the live ranges for data aggregates. Second, we have implemented this work in the SUIF and MachSUIF compilation framework. Our experimental results using a set of 10 C benchmarks from MediaBench and MiBench show that our methodology is capable of managing SPMs efficiently and effectively for large embedded applications. In addition, our SPM allocator can obtain close to optimal solutions when evaluated and compared against an existing heuristics-based SPM allocator and an ILP-based SPM allocator.

2. GRAPH COLORING SPM ALLOCATION

The basic idea is to formulate the SPM management problem as the classic register allocation problem. Figure 1 depicts an instantiation of our memory coloring methodology in the SUIF and MachSUIF compilation framework, where the four components of our methodology (highlighted in grey) are described below.

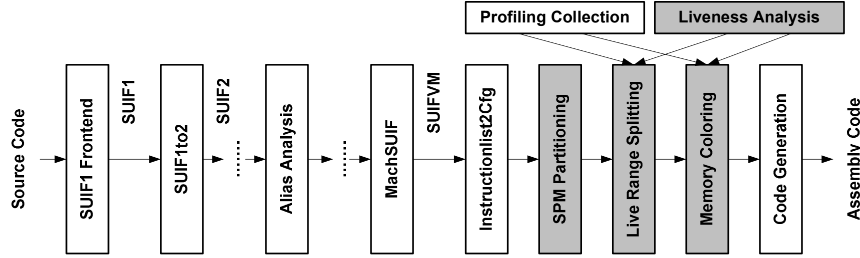

Fig. 1. An implementation of memory coloring in SUIF/MachSUIF.

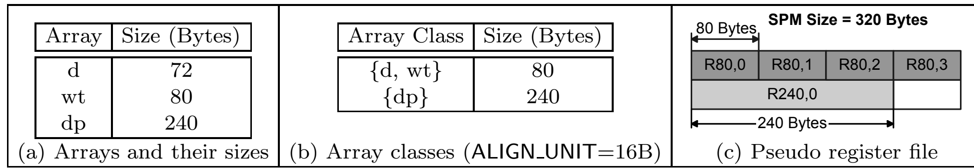

Fig. 2. An illustration of our SPM partitioning scheme.

In this work, we apply memory coloring to embedded programs that are often written in C or other C-like languages, in which arrays tend to be global or passed as parameters. Therefore, our instantiation is interprocedural by operating on one interference graph for a program.

Like garbage collectors, SPM allocators require some similar restrictions in programs, particularly those embedded programs written in C or C-like languages. Programming practices that disguise pointers such as casts between pointers and integers are forbidden. In addition, only portable pointer arithmetic operations on pointers and arrays are allowed. In general, C programs that rely on the relative positions of two arrays in memory are not portable. Indeed, comparisons (e.g., $<$ and $\leqslant$) and subtractions on pointers to different arrays are undefined or implementation-defined. Also, if n is an integer, $p \pm n$ is well-defined only if p and $p \pm n$ point to the same array. Fortunately, these restrictions are usually satisfied for static arrays and associated pointers in portable ANSI-compliant C programs.

2.1 SPM Partitioning

This component will partition the continuous space of an SPM into a pseudo register file (with interchangeable and aliased registers). This is the key to allowing us to map the SPM allocation problem into the classic register allocation problem.

The sizes of arrays in a program considered for SPM allocation are aligned to a pre-defined constant value, ALIGN_UNIT, in bytes. All arrays with a common aligned size are clustered into a common equivalent class called an *array class*. For each array class of a particular size, the SPM is partitioned into a register class such that each register in the class can hold exactly one array of that size. A detailed algorithm that formalizes one partitioning scheme can be found in [Li et al. 2005].

Figure 2 illustrates this partitioning scheme for a program if ALIGN_UNIT $= 16$ is assumed. For the three arrays given in Figure 2(a), the two array classes as given

in Figure 2(b) are obtained. Therefore, the SPM is partitioned into two register classes shown in Figure 2(c). Note that $R_{240,0}$ is aliased with $R_{80,0}$, $R_{80,1}$ and $R_{80,2}$.

2.2 Liveness Analysis

We have extended the traditional intraprocedural liveness analysis for scalars to compute the liveness information for arrays interprocedurally. Our analysis will be applied to a program twice, once before live range splitting to identify the liveness information to be used during the splitting process and once just before memory coloring to build the interference graph for the program.

The liveness analysis for arrays is conducted on the interprocedural CFG of a program (ICFG), which is constructed in the standard manner. By convention, the CFG of a function has a unique *entry block*, denoted ENTRY, and a unique *exit block*, denoted EXIT. In the CFG of each individual function, every call statement forms a basic block by itself and has a unique successor basic block called *return block*. The ICFG consists of the CFGs for all functions in the program and all possible interprocedural flow edges across the CFGs added conservatively as follows. At each call block (site), a directed edge is added from the call block to the ENTRY block of every possible callee function that may be invoked at the call site and a directed edge is added from the EXIT block of every such a callee function to the corresponding return block. In the presence of nonlocal control transfers such as `setjmp/longjmp` and exception handling, all possible interprocedural flow edges are also added to the ICFG. For example, exception propagation on the call stack from a callee to a caller results in additional interprocedural flow edges in the ICFG. The effect of these interprocedural flow edges is to force all data-flow analyses to safely approximate the control flow effects of these constructs.

The notion of liveness for arrays is interprocedural. An array is *live* at a program point if some of its elements may be used (or read) later on a control flow path in the ICFG of the program before all its elements are defined (or killed) on the path. The predicates, DEF and USED, local to a basic block $\mathcal{B}$ for an array A are defined as follows. $\mathsf{USED}_A(\mathcal{B})$ returns true iff some elements of A are read (possibly via pointers) in $\mathcal{B}$. $\mathsf{DEF}_A(\mathcal{B})$ returns true iff A is killed *entirely* in block $\mathcal{B}$, i.e., if every element of A is killed. In general, it is difficult to identify whether an array is killed or not at compile time. So we assume conservatively that an array that appears originally in a program is killed only at its definition block, i.e., the entry block of the scope where the array is defined. In addition, for every array copy statement introduced by Split_and_Copy in live range splitting (Section 2.3), the array that appears at its left-hand side is killed. Finally, for every edge connecting a call block and an ENTRY block, we assume the existence of a pseudo block $\mathcal{C}$ on the edge such that $\mathsf{DEF}_A(\mathcal{C})$ returns true iff A is neither global nor passed by a parameter at the corresponding call site and $\mathsf{USED}_A(\mathcal{C})$ always returns false. This makes our analysis context-sensitive since if A is a local array passed by a parameter in one calling context to a callee, then its liveness information obtained at that calling context will not be propagated into the others for the same callee function.

Given the above definitions of DEF and USED, the liveness information for an array A can be computed interprocedurally on the ICFG of the program by applying

the standard data-flow equations to the entry and exit of every block $\mathcal{B}$:

$$\begin{aligned} \mathsf{LIVEIN}_A(\mathcal{B}) &= (\mathsf{LIVEOUT}_A(\mathcal{B}) \wedge \neg\mathsf{DEF}_A(\mathcal{B})) \vee \mathsf{USED}_A(\mathcal{B}) \\ \mathsf{LIVEOUT}_A(\mathcal{B}) &= \bigvee_{\mathcal{S}\in\mathsf{succ}(\mathcal{B})} \mathsf{LIVEIN}_A(\mathcal{S}) \end{aligned} \tag{1}$$

where $\mathsf{succ}(\mathcal{B})$ denotes the set of all successor blocks of $\mathcal{B}$ in the ICFG.

2.3 Live Range Splitting

Live range splitting splits an array live range into several subranges, each of which can be assigned to a different pseudo register. In an embedded program, most of its execution time is spent in loops and most of its array accesses are made in loops. Therefore, we use an algorithm that is simple yet effective to split arrays only around frequently executed, i.e., hot loops. In particular, we only split the live range of an array in a loop if code rewriting required by splitting the array can be realized efficiently at the pre-header and exits of the loop.

A loop L is *splittable* only if L is reducible (in its containing function) and has (directly or indirectly) no nonlocal transfers such as `setjmp/longjmp` and exception-handling statements. This can be tested from the ICFG of the program. (L is not reducible in the presence of `setjmp` and exception-catching statements and the presence of `longjmp` and exception-throwing statements would complicate the code rewriting required in live range splitting.) Our experience indicates that the hot loops that appear in embedded C programs are generally splittable.

An array A accessed in a loop L that is contained in a function F is *splittable* if two conditions are met. First, if A is a global array, then A cannot be accessed in a function that may be called from both inside and outside L. Otherwise, A must be defined in F. Second, all pointers that may point to A in L are scalar pointers (which are pointers that point directly to A). This implies that A in L cannot be pointed to by fields in aggregates such as heap objects or arrays of pointers or indirectly by scalar pointers to pointers. As a result, the extra code inserted at the pre-header and exits of L due to splitting can be executed efficiently. In all embedded C benchmarks we have dealt with, static arrays are generally splittable.

Figure 3 gives an algorithm, Live_Range_Splitting, for splitting an array in a loop. All loops in a program are processed outside in when their containing functions in the call graph of the program are processed in topological order (lines 2 and 3). As a result, preference is given to split a global array at an outer loop. Recursion is rare in embedded C programs. So the call graph is simply made acyclic from a DFS traversal. Only splittable arrays are examined (line 4). An array A in a loop L will not be split again if it was done earlier in an enclosing loop (line 5).

In lines 8 - 17, we conduct a cost-benefit analysis to see if A can be split in L beneficially. Our cost model takes into account the access frequency of A and the data transfer cost between SPM and off-chip memory. The cost of communicating n bytes between SPM and off-chip memory is approximated by $C_s + C_t \times n$ (cycles), where C_s is the startup cost and C_t is the transfer cost per byte. M_{spm} (M_{mem}) denotes the cycle count required per element access to the SPM (off-chip memory).

The code rewriting procedure Split_and_Copy splits the live range of A in L contained in function F and rewrite L so that the split live ranges are all accessed

```
1   Procedure Live_Range_Splitting()
2   for every function F in the call graph used for building the ICFG of the program
        in topological order (make the call graph acyclic by removing the back edges
        in a DFS traversal on the call graph when recursion is found in the program)
3       for every splittable loop L in the loop hierarchy of function F in outside-in order
4           for every array A accessed and splittable inside L
5               if A has already been split in an enclosing loop of L in the ICFG continue
6               if SplitBenefit(F, L, A) ≤ SplitCost(F, L, A) continue
7               Split_and_Copy(F, L, A)

8   int Function SplitCost(F, L, A)
9   copy_freq_atPreheader = the execution frequency at the pre-header of L in function F
                            where a copy statement A' = A is inserted
10  copy_freq_atExits = the sum of execution frequencies at all exits of L in function F
                        where a copy statement A = A' is inserted at each exit
11  copy_freq = copy_freq_atPreheader + copy_freq_atExits
12  split_cost = (C_s + C_t × A.size) × copy_freq
13  return split_cost

14  int Function SplitBenefit(F, L, A)
15  access_freq = access frequency of A in L contained in function F
16  split_benefit = access_freq ×(M_mem − M_spm)
17  return split_benefit

18  Procedure Split_and_Copy(F, L, A)
19  Create a new array A' with the same aligned size as A
20  Add A' = A at the pre-header of L in function F
21  Let P_A be the set of scalar pointers that may point (directly) to A and accessed in L
22  for every pointer P in P_A
23      Add the following code at the pre-header of L in function F
          if P points to A
              Set offset_A = P−(base address of A)
              Set P to point to (base address of A' + offset_A)
24  Replace every direct access of A in L by an access to A'
25  if A may be modified in L
26      Add A = A' at every exit of L in function F where A is live
27  for every pointer P in P_A
28      Add the following code at every loop exit where P is live
          if P points to A'
              Set offset_A' = P−(base address of A')
              Set P to point to (base address of A + offset_A')
```

Fig. 3. An algorithm for splitting the live range of an array in a loop.

correctly. All accesses to A in L are redirected to A'. At the header of L, we add code to deal with every scalar pointer P that may (directly) point to A in L (lines 22 and 23). A runtime test is added to find out the unique target that P points to during program execution. If the target is A (identifiable easily at compile time since A is either global or local), then the offset of P relative to the beginning of A is dynamically computed. Then P is modified to point to the split live range A' at the same offset. At each loop exit (lines 27 and 28), P is restored to point to the correct position of the original array A if P is still live at the exit.

2.4 Memory Coloring

Given the register file and array candidates defined in SPM_Partitioning (including also the new arrays introduced by Live_Range_Splitting), this component determines which arrays should reside in which parts of the SPM by adapting an existing graph coloring algorithm for scalars that is generalized as described in [Smith et al. 2004] to deal with interchangeable and aliased registers. Instead of George and Appel's iterative-coalescing algorithm [George and Appel 1996] that we used earlier [Li et al. 2005], we have opted to use Park and Moon's optimistic-coalescing algorithm [Park and Moon 2004] since it is more aggressive in eliminating unnecessary live range splits and thus achieves better results in our experiments.

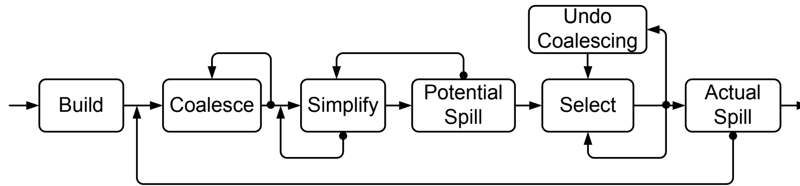

Fig. 4. Memory coloring via Park and Moon's optimistic-coalescing.

Figure 4 depicts the phase ordering of all phases in memory coloring using Park and Moon's optimistic-coalescing framework [Park and Moon 2004]. Below only some pertinent changes made to this framework are described.

In graph coloring register allocation, a live range is spilled to memory by inserting a load (store) instruction at every read (write) reference of the live range. At each of these insertion points, a new live range is introduced. As a result, the interference graph for a function needs to be rebuilt. In our case, an array candidate is spilled so that it will be accessed directly from the off-chip memory. No spill code and thus no new live ranges will ever be introduced. Therefore, the Build phase is only executed once in memory coloring and the interference graph for a program can be incrementally updated by simply removing the spilled live range from the graph. The updated interference graph will continue to be colored.

In the Coalesce phase, all split live ranges are coalesced. When a coalesced node cannot be colored in the Select phase, the Undo Coalescing phase will split it into a set of smaller live ranges so that some or all of these may be colored. All split live ranges that originate from a live range are treated as move-related nodes with its original live range. When some move-related nodes are coalesced, the corresponding data transfer operations will be eliminated accordingly.

3. EXPERIMENTAL RESULTS

Table I gives 10 embedded C benchmarks used in our experiments, where the first eight are from MediaBench and the last two from MiBench. The memory objects that are declared but not used in a benchmark are not counted. In our implementation depicted in Figure 1, all programs are compiled into assembly programs and then translated into binaries on a DEC Alpha 20264 architecture. The profiling information for MediaBench is obtained using the second data set available in the

Benchmark	#Lines	#Arrays	Array Data Set Size (KBytes)	Size of IG		Compile Time (secs)
				#Nodes	#Edges	
toast	6031	62	17.8	101	2750	0.244
untoast	6031	62	17.8	79	1787	0.221
rawcaudio	741	5	2.9	9	20	0.005
rawdaudio	741	5	2.9	9	20	0.005
pegwitencode	7138	121	226.7	230	17934	1.231
pegwitdecode	7138	121	226.7	224	17262	1.063
mpeg2encode	8304	62	9.2	108	1426	1.042
mpeg2decode	9832	76	21.8	101	1482	0.922
lame	18612	220	552.5	410	43863	4.299
rsynth	5713	75	44.6	91	3679	0.268

Table I. Benchmarks (where IG stands for interference graph).

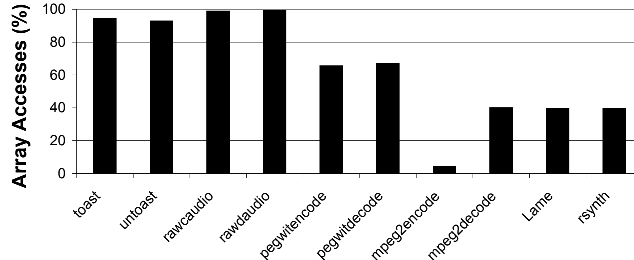

Fig. 5. Percentage of array accesses to all SPM-allocatable arrays over total memory accesses.

MediaBench website. These benchmarks are evaluated using the data sets that come with their source files. The profiling for the other benchmarks is obtained using inputs different from those when they are actually evaluated.

In embedded C programs, recursion is rarely used. All 10 benchmarks except `rsynth` are recursion-free. In `rsynth`, recursion is detected but no arrays are defined in any recursive function called directly or indirectly. Otherwise, a local array defined in a function can be callee-saved if it is found to be residing in SPM at a callee function. In addition, all associated pointers to the array to be redirected to the saved copy. These pointers should be scalar pointers for efficiency reasons.

In our experiments, ALIGN_UNIT is set to 16 bytes (Figure 2). We have compared the performance results obtained when ALIGN_UNIT takes four different values ranging from 8 bytes, 16 bytes, 32 bytes to 64 bytes. No significant performance variations are observed in all the 10 benchmarks used when ALIGN_UNIT is decreased from the default 16 bytes to 8 bytes. However, some performance slowdowns are observed in some benchmarks when ALIGN_UNIT is increased from 16 bytes to 32 bytes and 64 bytes, respectively. In these benchmarks, the negative impact on the utilization of the SPM space seems to be dominant. The positive impact on performance due to reduced register classes and aliases appears to be insignificant. Based on our experimental results, ALIGN_UNIT=16 appears to be a good default value (at least for the benchmarks used in our experiments).

We have modified SimpleScalar in order to carry out the performance evaluations for this work. There are four parameters involved in our cost model (Section 2.3).

Their values are set as $C_s = 100$, $C_t = 1$, $M_{\text{mem}} = 100$ and $M_{\text{spm}} = 1$.

In this work, we are concerned with assigning static data aggregates in a program to an SPM. So the scalars and heap objects in a program are ignored. However, our approach can be applied to deal with scalars in at least two ways. After register allocation for scalars has been performed, all spilled scalars are known. In one scheme, both memory-resident scalars (including spilled ones) are processed together with arrays by treating the scalars as a special case of arrays. In another scheme as described in Cooper and Harvey [1998], the memory-resident scalars can be assigned to a small portion reserved in the SPM. In our experiments, the accesses to all memory-resident scalars in `lame` represent only about 0.66% of its total memory accesses. This is the worst case among all benchmarks used. So only arrays are considered for SPM allocation in our experiments.

In `toast` and `untoast`, we have manually replaced a frequently used heap object with a global array so that it can be assigned to the SPM. For recursive functions in a program, we have extended memory coloring to handle the SPM-resident local arrays defined and used in recursive functions by using a callee-save mechanism.

Figure 5 shows the percentage of the array accesses to all arrays considered for SPM allocation over all memory accesses in a benchmark. For `toast`, `untoast`, `rawcaudio`, `rawdaudio`, `pegwitencode` and `pegwitdecode`, the majority of memory accesses are array accesses. For `mpeg2decode`, `lame` and `rsynth`, about 40% of all memory accesses are array accesses. There are varying reasons behind. For `rsynth`, more than 50% of the non-array memory accesses are generated by caller-callee register savings and some others are accesses to heap objects. For `mpeg2decode` and `lame`, the majority of the non-array memory accesses are made to heap objects. For `mpeg2encode`, its array accesses are not many since the most frequently accessed memory objects are heap objects rather than arrays.

Below we present and analyze the experimental results. In Section 3.1, we show that our approach is practically efficient. In Section 3.2, we evaluate its effectiveness in placing data in the SPM by comparing an SPM-based system with the one in which the SPM is not used. In Section 3.3, we compare our approach against two existing ones that rely on heuristics and integer linear programming (ILP), respectively. Our experimental results show that our SPM allocator can obtain close to optimal solutions for the 10 benchmarks used.

3.1 Compile Times

Table I gives the sizes of interference graphs and the average compile times (calculated across all SPM sizes considered on a 2.66GHz Pentium 4 with 2GB memory). These data suggest that our SPM allocator is efficient for all benchmarks used. Of all benchmarks, `lame` has the largest interference graph. Even in this worst case, the average compile time is only 4.299 secs.

3.2 Performance Improvements

Since we are concerned with placing arrays in SPM, we will evaluate the effectiveness of our approach in improving the utilization of SPM for all the arrays considered for SPM allocation. To this end, the concept of array hit rate is introduced. The *array hit rate* for a program is defined to be the percentage of array accesses hit in the SPM over the total array accesses considered for SPM allocation in the benchmark.

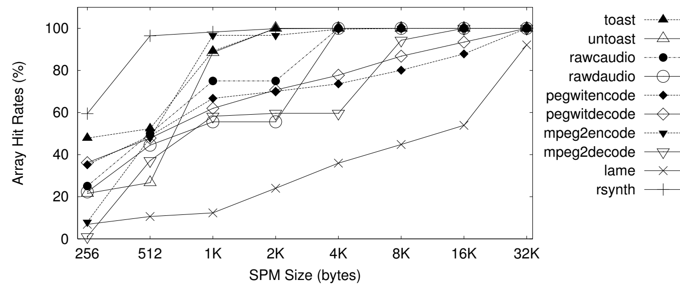

Fig. 6. Array hit rates.

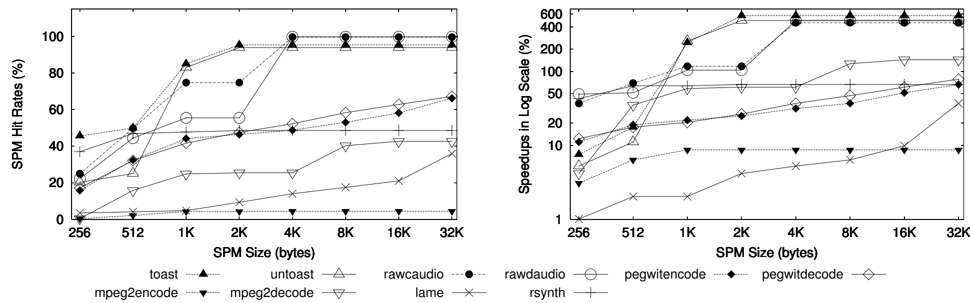

Fig. 7. Performance improvements on a system with SPM over the one without.

As in previous work [Kandemir et al. 2001; Avissar et al. 2002; Udayakumaran et al. 2006; Verma et al. 2004b], we will compare results obtained on a system incorporated with an SPM over the one in which the SPM is removed.

Figure 6 plots the array hit rates for the 10 benchmarks. For `toast`, `untoast`, `rawcaudio`, `rawdaudio` and `rsynth`, an SPM of 4K bytes is sufficient to hold all the arrays that are frequently accessed at any time during program execution. No significant hit rate increase is observed when a larger SPM is used. For `pegwitencode`, `pegwitdecode`, `mpeg2encode` and `mpeg2decode`, an SPM of 32K bytes is sufficient. As for `lame`, an SPM of 128K bytes is needed in order to keep all frequently accessed arrays in the SPM in all program regions. When the SPM size increases as shown in Figure 6, all the benchmarks exhibit non-decreasing array hit rate improvements. Each arrives at its peak at one of the SPM sizes used, where all its frequently accessed arrays can be found in the SPM throughout program execution.

The performance improvement of a program depends on (among others) the percentage of array accesses over the total memory accesses in the program, the SPM and memory access latencies and the DMA cost. Figure 7 gives the SPM, i.e., cache hit rates for all the benchmarks and the performance improvements of the benchmarks for the experimental settings described earlier. These results allow us to develop an informed understanding about the performance speedups achievable due to improved SPM hit rates. The best speedups (by a factor of over

400%) are achieved for `toast`, `untoast`, `rawcaudio` and `rawdaudio`. But only small performance improvements are observed for `mpeg2encode`.

The varying performance improvements across the 10 benchmarks can be understood by examining their array access percentages given in Figure 5 and their SPM hit rates given in Figure 7. For `mpeg2encode`, its array accesses represent only a small fraction of its memory accesses. So its performance speedups will not be impressive. When the SPM size increases as shown in Figure 7, the SPM hit rate for a benchmark keeps increasing until after the SPM size has exceeded a certain value. Then any further increase in the SPM size will have little positive impact on the SPM hit rate. As can be observed in Figure 5, the SPM hit rate for a program will eventually approach the array access percentage for the program, at which time all frequently accessed arrays in any program region are all found in the SPM.

The array copy costs between SPM and off-chip memory are less than 0.20% of the execution times for all benchmarks. In addition, optimistic-coalescing has successfully eliminated all array copy operations between pseudo registers.

3.3 Compared with Two Existing SPM Allocators

We compare our dynamic approach with two existing dynamic ones, which are drastically different in the sense that one resorts to integer linear programming (ILP) to search for optimal solutions and one relies on heuristics to obtain good solutions efficiently. The ILP-based approach, denoted ILP, is formulated based on the work described in [Verma et al. 2004b]. The heuristics-based approach, denoted HA, is the allocation scheme described in [Udayakumaran et al. 2006]. We present our results for ILP first in order to shed some light on the inherent time complexities of performing SPM allocation for our benchmark programs.

3.3.1 *Compared with ILP.*

An ILP-based approach for solving our SPM allocation problem *optimally* can be formulated in the standard manner. An example of how to do so for a slightly different SPM allocation problem can be found in [Verma et al. 2004b]. So only the key steps involved are explained below.

The ILP-based allocator is formulated to solve exactly the same problem as our memory coloring allocator. So the same live range splitting algorithm is applied. Linear constraints are introduced to keep track of which live ranges are in SPM or off-chip memory and whether a copy operation is needed in a move-related live range (due to splitting). Each live range is associated with an *offset variable* to identify its location in SPM if it happens to be assigned to SPM. For interfering live ranges, linear constraints are introduced to make sure that they will be assigned non-overlapping SPM spaces based on their associated offset variables. Finally, the objective function is to maximize the number of cycles saved on accessing the arrays (since they are assigned to SPM instead of off-chip memory) under consideration minus the number of cycles spent in array copy operations.

Table II presents the performance improvements that we can optimally expect from an ILP-based allocator over our memory coloring allocator. We used the commercial ILP solver, CPLEX 10.1, which is one of the fastest available in the market. By examining also Table I, we find that our allocator can achieve close to optimal results efficiently for all benchmarks across all configurations. Let us examine the performance results of ILP with respect to Table I and Figure 5. For `rawcaudio`

Benchmark	Speedups (%) under Eight Different SPM Sizes (Bytes)							
	256	512	1024	2048	4096	8192	16384	32768
toast	–	–	–	**0.1**/3m	**0.5**/2m	**0.0**/11m	**0.0**/31m	**0.0**/4m
untoast	–	–	–	**0.6**/21s	**0.2**/24s	**0.0**/4s	**0.0**/3s	**0.0**/3s
rawcaudio	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s
rawdaudio	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.4s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s	**0.0**/0.1s
pegwitencode	–	–	–	–	–	–	–	–
pegwitdecode	–	–	–	–	–	–	–	–
mpeg2encode	**0.0**/6s	**0.0**/1m	–	**0.1**/18h	**0.1**/15h	**0.1**/4h	**0.0**/1h	**0.0**/30m
mpeg2decode	–	**0.0**/1s	–	**0.0**/1m	**0.0**/1m	**0.0**/14s	**0.0**/14s	**0.0**/24s
lame	–	–	–	–	–	–	–	–
rsynth	–	–	–	–	**0.1**/18m	–	**0.3**/6m	**0.1**/22m

Table II. Performance improvements of ILP over memory coloring. For each configuration, X/Y means that ILP achieves an (optimal) speedup of X% over memory coloring with a solution time of Y on a 2.66GHz Pentium 4 with 2GB memory (where s stands for secs, m for mins and h for hours). A '–' for a configuration indicates that CPLEX cannot run to completion within 24 hours.

and `rawdaudio`, the number of arrays in each benchmark is small. So ILP terminates quickly and both allocators achieve the same results in all configurations. For `mpeg2encode` and `mpeg2decode`, the number of arrays in each benchmark is not as small but only a few of these are frequently accessed. In three configurations, ILP does not terminate. For the remaining ones, our allocator achieves nearly the same results as ILP. But ILP can take up to 18 hours to produce a solution that is only 0.1% better for `mpeg2encode` when the SPM size is 2K bytes. For `pegwitencode`, `pegwitdecode`, `lame` and `rsynth`, each has a large number of frequently accessed arrays, all of which must be explicitly dealt with in the ILP formulation. So ILP cannot run to completion in most of the configurations tested. Finally, for `toast` and `untoast`, ILP cannot terminate in the first few configurations.

In summary, ILP can yield optimal solutions efficiently in some configurations for certain benchmarks. However, its overall performance is unpredictable and may not run to completion within a given time limit. On the other hand, our allocator can obtain nearly optimal solutions efficiently in almost all cases.

3.3.2 *Compared with HA.* For the HA allocator introduced in [Udayakumaran et al. 2006], a set of heuristics are used to partition a program into a set of regions including procedures, loops, `if` statements and `switch` statements. A set of time stamps are assigned to the start point and end point of a region to represent the time(s) when the region is executed relative to the others. Then all the regions (their entries and exits to be precise) are processed according to the partial order of their time stamps. When each boundary point is processed, a set of heuristics are applied to determine which arrays will be copied to or evicted from SPM at that point. Finally, every SPM-resident array at a boundary point of a region is mapped to an SPM location. This may involve inserting array copy operations to compact, i.e., relocate the existing arrays in SPM at the point to create enough continuous memory blocks to store all SPM-resident arrays live at the point.

Figure 8 compares both allocators in terms of how effective they are in improving the array hit rate of a benchmark (defined in Section 3.2) and shows the resulting performance improvements of our approach over HA. For five benchmarks,

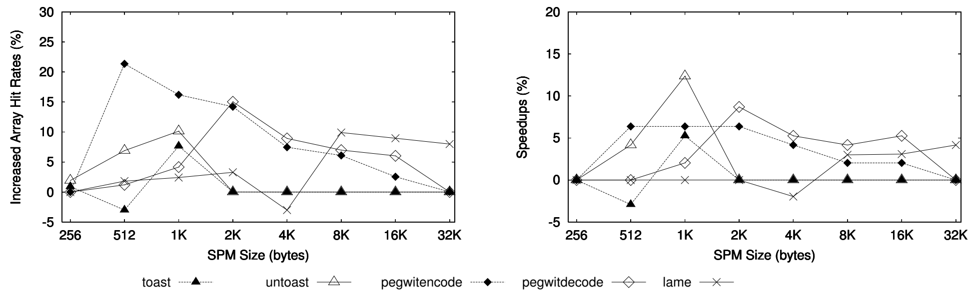

Fig. 8. Performance improvements of memory coloring over HA.

`rawcaudio`, `rawdaudio`, `mpeg2encode`, `mpeg2decode` and `rsynth`, both allocators achieve nearly the same array hit rates and thus nearly the same performance results in all SPM configurations tested. For the first four benchmarks, the reason behind is exactly the same as why our allocator and the ILP allocator also achieves nearly the same results in these cases, as explained in Section 3.3.1. For `rsynth`, both allocators happen to yield nearly the same performance results. So we discuss below only our experimental results for the remaining five benchmarks, `toast`, `untoast`, `pegwitencode`, `pegwitdecode` and `lame`.

As shown in Figure 8, our approach achieves better array hit rates than HA in most configurations for all the five benchmarks, which will translate into the performance speedups shown in the same graph. The largest performance improvement, a speedup of 13%, is achieved for `untoast` when the SPM size is 1K. This is possibly because HA may introduce array copy operations to compact SPM so as to create enough free blocks to store all SPM-resident arrays at a region entry or exit. On the other hand, memory coloring relies on optimistic coalescing to eliminate unnecessary live range splits and thus avoids unnecessary array copy operations. As discussed in Section 3.2, optimistic-coalescing has successfully eliminated all array copy operations within the SPM for all the 10 benchmarks used.

4. RELATED WORK

Earlier methods on assigning program data such as arrays or scalars [Avissar et al. 2002; Hiser and Davidson 2004; Sjödin and von Platen 2001; Steinke et al. 2002] to SPM are static in the sense that an array or scalar will reside either in SPM or in off-chip memory throughout program execution. In [Hiser and Davidson 2004], the authors provide an easily re-targetable compiler method for assigning data to many different types of memories. Steinke et al. [2002] propose a method that can place both data and code in SPM. In [Avissar et al. 2002; Sjödin and von Platen 2001], the static SPM allocation problem is formulated as an integer linear programming (ILP) program and the authors have shown that an optimal static SPM allocation scheme can be achieved for certain embedded applications.

Dynamic SPM allocation methods enable program data to be copied into and out of SPM during program execution. It has been demonstrated that a dynamic allocation scheme can often outperform an optimal static allocation scheme [Udayakumaran et al. 2006]. There are a few dynamic methods around [Kandemir et al.

2001; Udayakumaran et al. 2006; Verma et al. 2004b]. In [Kandemir et al. 2001], loop and data transformations are exploited but the proposed technique is applied to individual loop kernels in isolation. Udayakumaran et al. [2006] use a set of heuristics to guide their decision in deciding how to copy program data between SPM and off-chip memory during program execution. The ILP-based approach introduced in [Verma et al. 2004b] can yield optimal solutions for some programs but can be expensive when applied to others as reported in [Ravindran et al. 2005].

In [Udayakumaran et al. 2006; Steinke et al. 2002; Ravindran et al. 2005; Verma et al. 2004a], the authors show that it is also beneficial to place portions of program codes in SPM. In [Panda et al. 2000; 1997a; Verma et al. 2004a], researchers target a hybrid system with both cache and SPM. Therefore, their main objective is to place data in SPM to achieve better SRAM hit rates. In [Panda et al. 2000; 1997a; 1997b], solutions are proposed to map the variables that are likely to cause cache conflicts to SPM. In [Verma et al. 2004a], the authors propose a generic cache-aware scratchpad allocation algorithm to use scratchpad for storing instructions.

Graph coloring has been studied extensively in global register allocation [Chaitin 1982; Briggs et al. 1994; George and Appel 1996; Lueh et al. 2000; Park and Moon 2004]. Based on Chaintin's original formulation [Chaitin 1982], George and Appel [1996] introduced their well-known iterative-coalescing algorithm. Later, Smith et al. [2004] generalized this to handle irregular architectures with register classes and aliases, which is adopted here to assign data aggregates to an SPM. Genius [1998] applies graph coloring to reduce cross-interference cache misses caused by arrays accessed in loops.

The idea of live range splitting represents an important advance in the field of graph coloring register allocation. The live ranges of variables can be split into smaller pieces with copy instructions inserted to connect these pieces [Chow and Hennessy 1990; Appel and George 2001]. This may allow some or all smaller live ranges to be colored. A register allocator will typically be equipped with a coalescing phase to eliminate all redundant copies or moves introduced due to live range splitting. A number of coalescing approaches have been proposed [George and Appel 1996; Park and Moon 2004; Briggs et al. 1994]. In this work, we have relied on live range splitting to identify the program points at which arrays can be swapped in and out of an SPM. We have therefore relied on existing coalescing techniques to eliminate all redundant array copies (introduced due to live range splitting).

5. CONCLUSION

We have presented a new methodology for automatically assigning static data aggregates in a program to an SPM. The basic idea is to map the SPM management problem for data aggregates into the well-understood register allocation problem for scalars. We have instantiated our methodology in SUIF and MachSUIF and evaluated its usefulness in handling static arrays and structs for a set of 10 embedded C benchmarks from MediaBench and MiBench. Our experimental results show that memory coloring can solve the SPM management problem efficiently and effectively. In particular, by combining our splitting strategy and Park and Moon's optimistic-coalescing algorithm, we can obtain a practical compiler-directed ap-

proach to assigning static data aggregates in embedded applications to SPMs.

6. ACKNOWLEDGEMENT

The authors would like to thank all reviewers for their comments and suggestions. This work is supported by an Australian Research Council grant DP0881330.

REFERENCES

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

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

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

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

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

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

Genius, D. 1998. Handling cross interferences by cyclic cache line coloring. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques.* IEEE Computer Society, Washington, DC, USA, 112.

George, L. and Appel, A. W. 1996. Iterated register coalescing. *ACM Trans. Program. Lang. Syst. 18,* 3, 300–324.

Hiser, J. D. and Davidson, J. W. 2004. Embarc: an efficient memory bank assignment algorithm for retargetable compilers. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools.* ACM Press, 182–191.

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

Kapasi, U., Dally, W. J., Rixner, S., Owens, J. D., and Khailany, B. 2002. The Imagine stream processor. In *Proceedings 2002 IEEE International Conference on Computer Design.* 282–288.

Li, L., Gao, L., and Xue, J. 2005. Memory coloring: A compiler approach for scratchpad memory management. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques.* IEEE Computer Society, Washington, DC, USA, 329–338.

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

Panda, P. R., Dutt, N. D., and Nicolau, A. 1997a. Architectural exploration and optimization of local memory in embedded systems. In *ISSS '97: Proceedings of the 10th international symposium on System synthesis.* IEEE Computer Society, 90–97.

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

Panda, P. R., Dutt, N. D., and Nicolau, A. 2000. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Design Autom. Electr. Syst. 5,* 3, 682–704.

Park, J. and Moon, S.-M. 2004. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst. 26,* 4, 735–765.

RAVINDRAN, R. A., NAGARKAR, P. D., DASIKA, G. S., MARSMAN, E. D., SENGER, R. M., MAHLKE, S. A., AND BROWN, R. B. 2005. Compiler managed dynamic instruction placement in a low-power cod e cache. In *Proceedings of the 3rd IEEE/ACM International Symposium on Code G eneration and Optimization (CGO'03)*. 179–190.

SJÖDIN, J. AND VON PLATEN, C. 2001. Storage allocation for embedded processors. In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM Press, 15–23.

SMITH, M. D., RAMSEY, N., AND HOLLOWAY, G. 2004. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM Press, 277–288.

STEINKE, S., WEHMEYER, L., LEE, B., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, Washington, DC, USA, 409.

UDAYAKUMARAN, S., DOMINGUEZ, A., AND BARUA, R. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. on Embedded Computing Sys. 5,* 2, 472–511.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004a. Cache-aware scratchpad allocation algorithm. In *DATE'04: Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, Washington, DC, USA, 21264.

VERMA, M., WEHMEYER, L., AND MARWEDEL, P. 2004b. 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*. ACM Press, New York, NY, USA, 104–109.

WOLFE, M. 1989. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 357–361.

XUE, J. 2000. *A Loop Tiling Theory for Optimizing Compilers*. Kluwer Academic Publishers.