# Improving Scratchpad Allocation with Demand-Driven Data Tiling

Xuejun Yang
School of Computer
National University of Defense Technology
Changsha, China
xjyang@nudt.edu.cn

Li Wang
School of Computer
National University of Defense Technology
Changsha, China
dragonylffly@163.com

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
jingling@cse.unsw.edu.au

Tao Tang
School of Computer
National University of Defense Technology
Changsha, China
tt.tang84@gmail.com

Xiaoguang Ren
School of Computer
National University of Defense Technology
Changsha, China
hbszrxg@gmail.com

Sen Ye
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
longquan_135@126.com

## ABSTRACT

Existing scratchpad memory (SPM) allocation algorithms for arrays, whether they rely on heuristics or resort to integer linear programming (ILP) techniques, typically assume that every array is small enough to fit directly into the SPM. As a result, some arrays have to be *spilled entirely* to the off-chip memory in order to make room for other arrays to stay in the SPM, resulting in sometimes poor SPM utilization.

In this paper, we introduce a new comparability graph coloring allocator that integrates data tiling and SPM allocation for arrays by tiling arrays on-demand to improve utilization of the SPM. The basic idea is to repeatedly identify the heaviest path in an array interference graph and then reduce its weight by tiling certain arrays on the path appropriately with respect to the size of the SPM. The effectiveness of our allocator, which is presently restricted to tiling 1-D arrays, is validated by using a number of small benchmark kernels for which existing allocators are ineffective (if tiling is not applied). More sophisticated tiling heuristics appropriate for demand-driven data tiling, once devised, are expected to be useful for improving utilization of SPM for whole-program applications.

## Categories and Subject Descriptors

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

## General Terms

Algorithms, Experimentation, Languages, Performance


*CASES'10,* October 24–29, 2010, Scottsdale, Arizona, USA.


## Keywords

Scratchpad memory, software-managed cache, comparability graph coloring, data tiling, loop tiling

## 1. INTRODUCTION

Hardware-managed cache has traditionally been used to bridge the ever-widening performance gap between processor and memory. Despite this great success, some deficiencies with cache are well-known. First, their complex hardware logic incurs high overhead in power consumption and area. Second, their simple application-independent management strategy does not benefit from some data access characteristics in many applications. Finally, their uncertain access latencies make it difficult to guarantee real-time performance in real-time applications.

In contrast, software-managed scratchpad memory (SPM) has advantages in power, area, real-time guarantees and performance [1]. Thus, SPM is widely adopted in embedded systems, stream architectures (known as stream register file, local memory or streaming memory), and GPUs (known as shared memory in NVIDIA GPUs under its CUDA programming model). In the case of supercomputers, software-managed on-chip memory is also frequently used, especially in their accelerators. Examples include Merrimac [5], Cyclops64 [4], Grape-DR [17] and Roadrunner [2].

Unlike cache-based machines, machines with SPMs require software to explicitly and carefully manage data allocation in the SPM and make fully use of the scarce on-chip memory. Manual SPM management is impractical and error-prone, leading to non-portable code.

Many compiler approaches, static or dynamic, for SPM allocation have been proposed. Dynamic approaches, which allow arrays to be swapped into and out of SPM during run time, are known to outperform their static counterparts. However, the proposed dynamic SPM allocators typically assume that every array candidate is small enough to fit directly into the SPM. As a result, some arrays have to be *spilled entirely* to the off-chip memory to make room for other arrays to stay in the SPM, resulting in sub-optimal solutions.

Data tiling [12, 10], which partitions a large array into smaller subarray tiles, was originally proposed to improve the cache performance of regular loop kernels, i.e., loop kernels whose data dependences are mostly constant or uniform. This data transforma-

tion technique was later applied to improve utilization of SPM by copying the subarray tiles of an array, one at a time, rather than the entire array itself between SPM and off-chip memory [11, 16]. While being effective for certain programs, these earlier methods exhibit some deficiencies as discussed in Section 2.

In this paper, we introduce a new comparability graph coloring allocator that integrates for the first time data tiling and SPM allocation for arrays by tiling arrays on-demand to improve utilization of the SPM. Central to graph coloring is the notion of *array interference graph*. Given a program, its array interference graph $\mathcal{G}$ consists of the nodes representing all the arrays in the program and the edges between two nodes representing the fact that the two nodes have overlapping live ranges (i.e., code regions) and thus cannot be placed in overlapping spaces in the SPM.

We propose to solve the SPM allocation problem for $\mathcal{G}$ by first completing it into a comparability graph $\mathcal{G}'$ and then finding an optimal acyclic orientation $\alpha$ for $\mathcal{G}'$. As a result, the SPM space required by $\mathcal{G}$ is bounded from below by the heaviest (directed) path $\mathcal{P}$ in the directed graph of $\mathcal{G}'$ induced by $\alpha$. This observation motivates us to perform data tiling on-demand during the iterative scratchpad allocation process (common in a graph coloring allocator). The novelty of our approach lies in identifying the heaviest directed path this way during each iteration step and reducing its weight (if it is still larger than the SPM size) by tiling certain arrays on the path appropriately with respect to the size of the SPM under consideration. The optimal tile sizes required can be found either analytically or numerically with a cost-benefit analysis. In the case when data tiling is not possible, some arrays are spilled entirely to reduce the weight of $\mathcal{P}$.

This paper makes the following contributions:

- We present a new SPM allocator that combines data tiling and scratchpad allocation to improve utilization of SPM.
- We propose to perform demand-driven data tiling during graph coloring scratchpad allocation so that tile sizes can be found with a cost-benefit analysis.
- We demonstrate the effectiveness of our SPM allocator by using a number of benchmark kernels for which existing allocators are ineffective (if tiling is not applied).

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces some basic results required to understand our approach. Section 4 presents our algorithm. Section 5 evaluates our approach. Section 6 concludes the paper.

# 2. RELATED WORK

Existing approaches for SPM allocation are either static or dynamic. The static approaches are known to be less efficient than dynamic counterparts. The proposed dynamic approaches can be roughly divided into two classes, those that resort to ILP [21] and those that rely on some well-crafted heuristics [11, 19, 20, 14, 15, 13]. The ILP-based approaches are theoretically optimal but too expensive to be practical for many applications. Among the heuristics-based approaches, graph coloring [14] seems to achieve the best performance for general-purpose applications [13] with interval coloring delivering better performance for embedded applications [15]. However, these existing SPM allocators always spill an array entirely whenever the SPM space is insufficient, resulting in sub-optimal solutions.

Kandemir et al. [11], Zhang and Kurdahi [23] and Li et al. [16] apply data tiling [12] to improve utilization of SPM. However, the methods described in [11, 23] are restricted to the matrix multiplication kernel only while [16] relies on ILP to find optimal tile sizes to tile user-specified arrays in an ILP-based allocator.

Fabri [6] discovered the connection between interval coloring and compile-time memory allocation. Li et al. [15] apply interval coloring to assign arrays in embedded programs to SPM. Yang et al. [22] apply comparability graph coloring to optimize utilization of the stream register file in a stream architecture, with the assumption that every stream candidate (i.e., array) is small enough so that it can be placed entirely in a stream register file.

# 3. BACKGROUND

This section recalls some basic results about interval coloring and comparability graph coloring from [7] as well as minimal comparability completion from [9], providing a basis for understanding our proposed approach.

## 3.1 Interval Coloring vs. SPM allocation

The SPM allocation problem can be naturally solved by interval coloring as formulated below. Allocating SPM spaces to array live ranges in an array interference graph, IG, is represented by an assignment of intervals to the nodes in the IG. Minimizing the span of intervals amounts to minimizing the required SPM size.

DEFINITION 1. *Given an IG $\mathcal{G} = (V, E)$ with positively integral node weights $w : V \to \mathbb{N}$ (representing array sizes), an* interval coloring *$\alpha$ of $\mathcal{G}$ maps each node $x$ onto an interval $\alpha_x$ of a real line of width $w(x)$ so that adjacent nodes are mapped to disjoint intervals, i.e., $(x, y) \in E$ implies $\alpha_x \cap \alpha_y = \emptyset$.*

Given an undirected graph $\mathcal{G} = (V, E)$ with the function $w$ mapping nodes to positively integral weights, the total *width* of an interval coloring $\alpha$, $\chi_\alpha(\mathcal{G}; w)$, is $|\bigcup_{x \in V} \alpha_x|$. The *chromatic number* $\chi(\mathcal{G}; w)$ is the smallest width used to color the nodes in $\mathcal{G}$, which corresponds to the optimal SPM allocation.

## 3.2 Interval Coloring vs. Acyclic Orientation

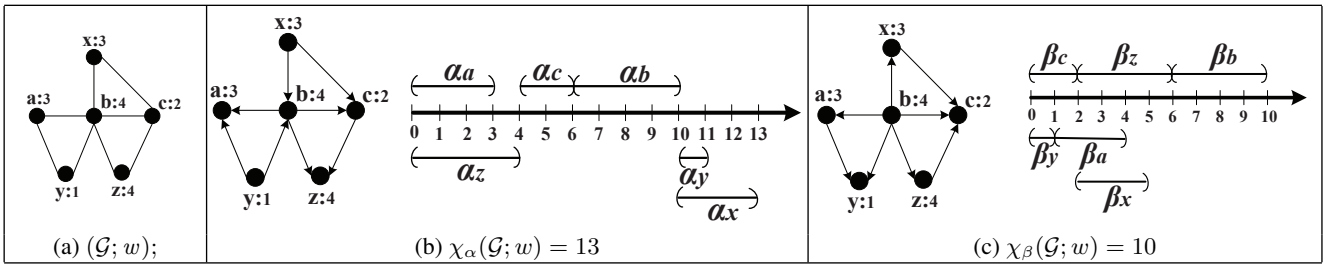
Let $\mathcal{G}= (V, E)$ be an undirected graph. The subgraph of $\mathcal{G}$ induced by a subset $V' \subseteq V$ of nodes in $\mathcal{G}$ is denoted by $\mathcal{G}[V']$. An *orientation* of $\mathcal{G}$ is a function $\alpha$ that assigns every edge a direction such that $\alpha(x, y) \in \{(x,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

Every interval coloring $\alpha$ of $\mathcal{G}$ induces an acyclic orientation $\alpha'$ such that $(x, y) \in \alpha'$ if and only if $\alpha_x$ is to the right of $\alpha_y$ for all $(x, y) \in E$ (by Definition 1). Conversely, an acyclic orientation $\alpha$ of $\mathcal{G}$ induces an interval coloring $\alpha'$. This can be achieved as follows. For a sink node $x$ (without successors), let $\alpha'_x = [0, w(x))$. Proceeding inductively, for a node $y$ with all its successors already colored, let $\alpha'_y = [t, t + w(y))$, where $t$ is the largest endpoint of their intervals.

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

$$\chi(\mathcal{G}; w) \quad = \quad \min_{\alpha \in \mathcal{A}(\mathcal{G})} \Big( \max_{\mu \in \mathcal{P}(\alpha)} \mathrm{w}(\mu) \Big) \tag{1}$$

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




(a) $(\mathcal{G}; w)$; (b) $\chi_\alpha(\mathcal{G}; w) = 13$ (c) $\chi_\beta(\mathcal{G}; w) = 10$

**Figure 1: Two interval colorings $\alpha$ and $\beta$ of a weighted undirected graph and their acyclic orientations.**

Figure 1 illustrates the equivalence between finding an interval coloring and finding an acyclic orientation for a weighted graph. In Figure 1(b), the heaviest path is $x \rightarrow b \rightarrow c \rightarrow z$ with a (total) weight of $\chi_\alpha(\mathcal{G}; w) = 13$. In Figure 1(c), the heaviest path is $b \rightarrow z \rightarrow c$ with a weight of $\chi_\beta(\mathcal{G}; w) = 10$. The gap between the two is 3 but can be larger in general. So there is a need to look for an optimal solution efficiently in practice.

## 3.3 Comparability Graph Coloring

For the purposes of optimizing utilization of SPM, we examine below a class of graphs that allows interval colorings to be found optimally in polynomial time.

DEFINITION 2. *An orientation $\alpha$ of an undirected graph $\mathcal{G}$ is* transitive *if $(x, z) \in \mathcal{G}_\alpha$ whenever $(x, y), (y, z) \in \mathcal{G}_\alpha$.*

DEFINITION 3. *An undirected graph $\mathcal{G}$ is a* comparability graph *if there exists a transitive orientation of $\mathcal{G}$.*

A transitive orientation is acyclic but the converse is not necessarily true. In Figure 1(b), $\alpha$ is not transitive since $(y, b), (b, c) \in \mathcal{G}_\alpha$ but $(y, c) \notin \mathcal{G}_\alpha$. However, $\beta$ shown in Figure 1(c) is transitive. As a result, the graph given in Figure 1(a) is a comparability graph.

THEOREM 1. *For any transitive orientation $\alpha$ of $\mathcal{G}$, the interval coloring induced for $\mathcal{G}$ is optimal.*

Furthermore, the problems of recognizing a comparability graph $\mathcal{G} = (V, E)$ and finding a transitive orientation of $\mathcal{G}$ can both be done in $O(\delta \cdot \mid E \mid)$ time and $O(\mid V \mid + \mid E \mid)$ space, where $\delta$ is the maximum of the degrees of all nodes in $\mathcal{G}$. Based on $\alpha$, an optimal coloring of $\mathcal{G}$ can be obtained in linear time [7].

## 3.4 Minimal Comparability Completion

Given a graph $\mathcal{G}$, a comparability graph obtained by adding edges to $\mathcal{G}$ is called a *comparability completion* of $\mathcal{G}$. Computing a comparability completion of $\mathcal{G}$ with the minimum number of added edges (called a *minimum comparability completion*) is an NP-hard problem [8]. As an approximation to the minimum comparability completion, a *minimal comparability completion* $\mathcal{H}$ of $\mathcal{G}$ is a comparability completion of $\mathcal{G}$ such that no proper subgraph of $H$ is a comparability completion of $\mathcal{G}$. Obviously, a minimum comparability completion is minimal but the converse is not necessarily true. A polynomial algorithm is presented in [9] to compute a minimal comparability completion for a graph $\mathcal{G}$.

In our SPM allocator described below, a given IG may be subject to the minimal comparability completion in order to obtain improved SPM utilization.

# 4. SPM ALLOCATION WITH ON-DEMAND DATA TILING

Our SPM allocator, given in Figure 2, performs data tiling on-demand in order to improve utilization of the SPM. Presently, our algorithm is restricted to tiling 1-D arrays. Higher-dimensional arrays can be converted to 1-D arrays if it is desirable for them be tiled. Our SPM allocator proceeds in five main steps, which are described in five separate subsections.

- **Live-Range Splitting** (Section 4.1). As in [14, 13], the live ranges of the arrays inside loop nests are split based on a cost-benefit analysis. The new live ranges obtained inside loop nests are called *hot arrays* as they are frequently accessed. Copy operations are inserted at the splitting points to transfer the *hot arrays* between SPM and off-chip memory.

- **Finding the Critical Path** (Section 4.2). Next, the IG $\mathcal{G}_o$ for the program is built. Its subgraph containing all hot arrays is completed into a comparability graph $\mathcal{H}_t$ (if necessary). Then a transitive orientation $\alpha$ of $\mathcal{H}_t$ is computed, from which the critical path, i.e., heaviest path $\mathcal{P}_{\mathcal{H}_t}$ is deduced.

- **Live-Range Coalescing** (Section 4.3). If the weight of the critical path $\mathcal{P}_{\mathcal{H}_t}$ is not larger than the SPM size, then all array candidates in $\mathcal{H}_t$ can be placed in the SPM. The algorithm tries to coalesce live ranges to eliminate unnecessary copy operations introduced during live range splitting before terminating.

- **Spilling** (Section 4.4). If the weight of $\mathcal{P}_{\mathcal{H}_t}$ exceeds the SPM size, then either spilling or data tiling is applied. But the latter is preferred since it often tends to make a better use of the scarce SPM space.

- **Data Tiling** (Section 4.5). The arrays on the critical path $\mathcal{P}_{\mathcal{H}_t}$ are tiled on-demand to reduce its weight so that better SPM utilization may be obtained. Guided by $\mathcal{P}_{\mathcal{H}_t}$ and a cost benefit analysis, an optimal tile size is determined in a symbolic manner.

The main contribution of this paper is a graph-coloring-based scratchpad allocation method for data aggregates that performs both scratchpad allocation and data tiling together. Due to the iterative nature of graph coloring register/scratchpad allocation [3, 14], various heuristics such as those for live range splitting and spilling are employed. We use mostly existing heuristics with some slight variations, if necessary. Other more effective heuristics can be developed in future.

1: **procedure** spm_alloc
2: Live_Range_Splitting()
3: $\mathcal{G}_o = (V_o, E_o)$ = Build_Interference_Graph();
4: Let $V_t$ be the set of introduced arrays, i.e., hot arrays during live range splitting
5: Let $\mathcal{G}_t = \mathcal{G}_o[V_t]$ // subgraph induced by $V_t$
6: **if** $\mathcal{G}_t$ is a comparability graph **then**
7: Let $\mathcal{H}_t = \mathcal{G}_t$
8: **else**
9: Let $\mathcal{H}_t$ be a minimal comparability completion of $\mathcal{G}_t$
10: **end if**
11: Let $\alpha$ be a transitive orientation of $\mathcal{H}_t$
12: Let $\mathcal{P}_{\mathcal{H}_t}$ be the heaviest directed path in $\alpha$
13: Let $w(\mathcal{P}_{\mathcal{H}_t})$ be the sum of weights of nodes in $\mathcal{P}_{\mathcal{H}_t}$
14: **if** $w(\mathcal{P}_{\mathcal{H}_t}) \leq SPM\_Size$ **then**
15: Goto 29
16: **end if**
17: Let $TAS_{\mathcal{P}_{\mathcal{H}_t}}$ be the set of tileable arrays in $\mathcal{P}_{\mathcal{H}_t}$
18: **if** $TAS_{\mathcal{P}_{\mathcal{H}_t}} == \emptyset$ **then**
19: **while** $w(\mathcal{P}_{\mathcal{H}_t}) > SPM\_Size$ **do**
20: Choose a node $v$ from $\mathcal{P}_{\mathcal{H}_t}$ to spill
21: $\mathcal{H}_t = \mathcal{H}_t[V_t - \{v\}]$
22: **end while**
23: Goto 12
24: **end if**
25: Choose a loop $\mathcal{L}$ which accesses some or all nodes in $TAS_{\mathcal{P}_{\mathcal{H}_t}}$
26: Undo the live range splitting for $\mathcal{L}$ and eliminate all hot arrays introduced earlier for $\mathcal{L}$
27: Perform a loop/data tiling to $\mathcal{L}$ with the tile size set as $x$ = Determine_Optimal_Tile_Size($\mathcal{L}, \mathcal{P}_{\mathcal{H}_t}$)
28: Goto 2
29: $\mathcal{G}_t$ = Coalesce_Live_Ranges($\mathcal{G}_o, \mathcal{G}_t$)
30: Let $\alpha$ be a transitive orientation of $\mathcal{G}_t$
31: **Output:** $\alpha$
32: **end procedure**

**Figure 2: SPM Allocation with On-Demand Tiling.**

## 4.1 Live Range Splitting

An array may be frequently accessed at some parts of its live range, i.e., at some computation-intensive loops. Following [13], we split its live range around loops and insert the required array copy operations at the splitting points, which become potentially the data transfer statements between the SPM and off-chip memory. For a loop nest where an array is accessed, the array is copied to a new, i.e., hot array at the earlier splitting point (at the beginning of the loop nest) and restored back at the later splitting point (at the end of the loop nest). During our graph coloring stage, all these hot arrays are the candidates to be colored so that they will likely be placed in the SPM.

The live ranges of arrays in a program are required in order to perform live range splitting and construct an IG for the program later. The live ranges of arrays are computed by extending the def/use definitions for scalars to arrays in the normal manner. At any program point, USE($\mathcal{A}$) returns true iff some elements of $\mathcal{A}$ are read. DEF($\mathcal{A}$) returns true iff $\mathcal{A}$ is defined entirely, i.e., if every element of $\mathcal{A}$ is defined. In general, it is difficult to identify whether an array is defined or not at compile time. So we assume conservatively that an array that appears originally in a program is defined only at its definition point, i.e., where it is declared. In addition, for every array copy introduced in live range splitting, the array that appears at its left-hand side is defined. The live range of an array starts from its definition and ends at its last use. Two arrays are *move-related* if one is obtained as a result of splitting the live range of the other. Such move-related arrays can be coalesced if the corresponding splits are unnecessary.

Consider our algorithm in Figure 2. In line 2, live range splitting is applied to the program under consideration. We adopt the algorithm described in [13] to perform live range splitting. This algorithm processes all the loop nests in every function one by one and examines all the loops of a particular loop nest, starting from its outermost to innermost loop. For every array $\mathcal{A}$ accessed in a loop $\mathcal{L}$, the algorithm checks to see if it is beneficial to split the live range of $\mathcal{A}$. The cost model takes into account the access frequencies of arrays (obtained by compiler analysis as well as 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 approximated by $C_s + C_t \times n$ (cycles), where $C_s$ is the startup cost and $C_t$ the transfer cost per byte. In addition, $S_{\text{spm}}$ and $S_{\text{mem}}$ are used to represent the number of cycles required per array element access to the SPM and off-chip memory, respectively. If the splitting is beneficial, then a new array is introduced and appropriate copy in/out operations are inserted.

Consider an example program in Figure 3, which will be used to illustrate our SPM allocation algorithm. Let $C_s = 90$, $C_t = 10$, $S_{\text{spm}} = 1$ and $S_{\text{mem}} = 100$. Consider $d$, which is frequently accessed in two loops, $\mathcal{L}_1$ in lines 4 – 6 and $\mathcal{L}_2$ in lines 9 – 11. Let us examine $\mathcal{L}_1$ first. The access frequency of $d$ is $16 * 4 = 64$ bytes (if one float is 4-byte long). The split benefit is $64 * (100 - 1) = 6336$ while the split cost is $(90 + 10 * 26 * 4) * 2 = 2260$. As a

```
1    float a[32], b[16],  c[48], d[26];
2    int i, sum;
3    ...
4    for (i = 1:16) {
5       b[i] = b[i] + a[2i]*c[3i] + d[i];
6    }
7    ...
8    sum = 0;
9    for (i = 1:16) {
10      sum = sum + b[i]*d[i+10];
11   }
12   data_save(sum);
```

**Figure 3: An example program.**

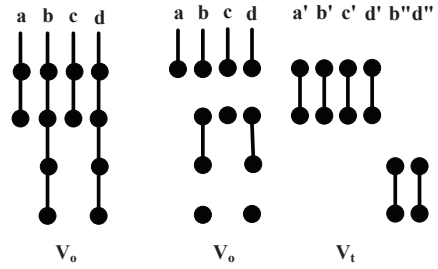
result, it is beneficial to perform live range splitting. The live range splitting for $d$ in loop $\mathcal{L}_2$ and for other arrays is done similarly. The final program is given in Figure 4. The live ranges before/after the splitting step are shown in Figure 5.

```
1    float a[32], b[16], c[48], d[26];
2    int i, sum;
3    ...
4    float a'[32], b'[16], c'[48], d'[26];
5    copy(a, a');
6    copy(b, b');
7    copy(c, c');
8    copy(d, d');
9    for (i = 1:16) {
10      b'[i] = b'[i] + a'[2i]*c'[3i] + d'[i];
11   }
12   copy(b', b);
13   copy(d', d);
14   ...
15   sum = 0;
16   float b"[16], d"[26];
17   copy(b, b");
18   copy(d, d");
19   for (i = 1:16) {
20      sum = sum + b"[i]*d"[i+10];
21   }
22   data_save(sum);
```

**Figure 4: The program of Figure 3 after splitting.**

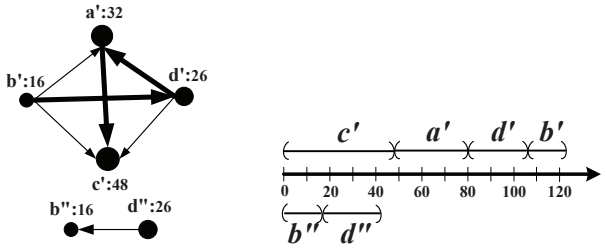
## 4.2 Finding the Critical Path

Let us continue our discussion with our algorithm in Figure 2. In line 3, the IG $\mathcal{G}_o$ for all array live ranges is built. Among all the live ranges, the ones introduced in $V_t$ during live range splitting are expected to be placed in SPM. So the interference subgraph induced by them, $\mathcal{G}_t$, is extracted from $\mathcal{G}_o$ (line 5). In line 6, the algorithm checks to see if $\mathcal{G}_t$ is a comparability graph, which holds in many applications, since the array IGs tend to be disjoint cliques (complete graphs), which are trivially comparability graphs. If $\mathcal{G}_t$ is not a comparability graph by itself, the minimal comparability completion algorithm is performed to make it so (line 9). Next, a transitive orientation $\alpha$ of the comparability graph is attained (line 11), and from which the heaviest directed path $\mathcal{P}_{\mathcal{H}_t}$ is derived (line 12). The transitive orientation $\alpha$ corresponds to an optimal interval coloring of $\mathcal{H}_t$. Thus, $\mathcal{P}_{\mathcal{H}_t}$ is actually the critical path that determines precisely the amount of SPM space required by $\mathcal{H}_t$.




(a) Before (b) After

**Figure 5: Live ranges before/after splitting.**




(a) $\mathcal{G}_t$ (b) Coloring

**Figure 6: Interference subgraph and its coloring.**

For the program in Figure 4, the interference subgraph $\mathcal{G}_t$ and its transitive orientation are depicted in Figure 6(a). With two disjoint cliques, $\mathcal{G}_t$ is trivially a comparability graph. The heaviest directed path is highlighted in thick arrows, $b' \rightarrow d' \rightarrow a' \rightarrow c'$, with a total weight of 122, which gives a lower bound for the SPM capacity to fully hold all these live ranges. The optimal coloring, i.e., the optimal allocation corresponding to the orientation is shown in Figure 6(b).

## 4.3 Live Range Coalescing

If the weight of $\mathcal{P}_{\mathcal{H}_t}$ is not larger than the SPM size (line 14 in Figure 2), then the current candidates in $\mathcal{H}_t$ can all be placed in the SPM. In this case, we apply coalescing to remove unnecessary copy operations (if any) introduced during live range splitting. However, coalescing may increase SPM pressure and is thus performed with the IG being always colorable (line 29).

The move-related nodes that are not originally in $\mathcal{G}_t$ are inserted back into $\mathcal{G}_t$ and coalesced by applying Coalesce_Live_Ranges in line 29 using the optimistic coalescing algorithm [18]. Every time after some coalescing has been done, the current graph is checked to see if it remains a comparability graph. If it is not, the minimal comparability graph completion is performed to make it so. Next, the heaviest path $\mathcal{P}_{\mathcal{H}_t}$ is recalculated. If $w(\mathcal{P}_{\mathcal{H}_t}) \leq SPM\_Size$, the coalescing results are kept, otherwise discarded, until all coalescing possibilities have been tried. Finally, a transitive orientation to the final IG, i.e., a SPM allocation is returned (line 31).

For example, Figure 7(a) depicts the IG $\mathcal{G}_o$ corresponding to the program in Figure 4 with two copy-related nodes $b$ and $d$ being inserted. Figure 7(b) shows the graph after the two move-related pairs $(b, b")$ and $(d, d")$ are coalesced. Figure 7(c) shows the graph after

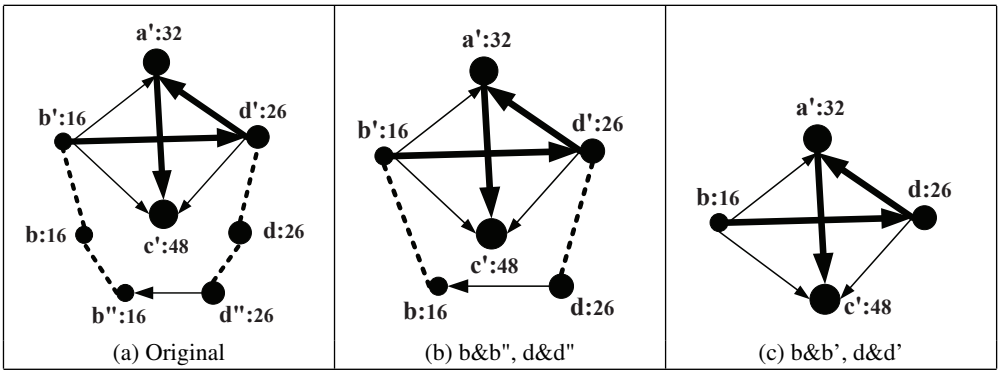



**Figure 7: Live range coalescing.**

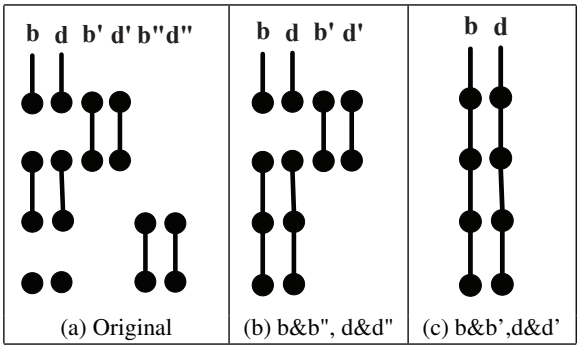



**Figure 8: Effects of coalescing on live ranges.**

the two more move-related pairs $(b, b')$ and $(d, d')$ are further coalesced. Figure 8 illustrates intuitively the effects of these coalescing steps on live ranges.

```
1    float a[32], b[16], c[48], d[26];
2    int i, sum;
3    ...
4    float a'[32], c'[48];
5    copy(a, a');
6    copy(c, c');
7    for (i = 1:16) {
8      b[i] = b[i] + a'[2i]*c'[3i] + d[i];
9  }
10   ...
11   sum = 0;
12   for (i = 1:16) {
13      sum = sum + b[i]*d[i+10];
14   }
15   data_save(sum);
```

**Figure 9: The program after live range coalescing.**

Figure 9 gives the program after live range coalescing is performed. Compared to the original program in Figure 4, six copy operations have been eliminated.

## 4.4 Spilling

If the weight of $\mathcal{P}_{\mathcal{H}_t}$, $w(\mathcal{P}_{\mathcal{H}_t})$, is larger than the SPM size, we resort to spilling or data tiling to reduce the weight of $\mathcal{P}_{\mathcal{H}_t}$. We prefer data tiling since doing so enables more live ranges to be placed in the SPM, resulting in often more significantly improved SPM utilization.

Let us return to our algorithm in Figure 2. In line 17, the set of tileable arrays $TAS_{\mathcal{P}_{\mathcal{H}_t}}$ is extracted from $\mathcal{P}_{\mathcal{H}_t}$. If no array can be tiled (line 18), spilling has to be performed. We adopt the heuristic introduced in [15] except that we find the heaviest path rather than (maximum) cliques related to $\mathcal{H}_t$ so that the resulting spilling (and data tiling) phases are polynomial.

The *colorability* of $\mathcal{H}_t$ is governed by:

$$\alpha(\mathcal{H}_t) = \frac{SPM\_SIZE}{\max(SPM\_SIZE, w(\mathcal{P}_{\mathcal{H}_t}))} \times \mathcal{P}_{\mathcal{H}_t}.freq \qquad (2)$$

where $\mathcal{P}_{\mathcal{H}_t}$ is the heaviest path of an transitive orientation of $\mathcal{H}_t$ and $\mathcal{P}_{\mathcal{H}_t}.freq$ is the sum of the access frequencies of all arrays in $\mathcal{P}_{\mathcal{H}_t}$. Intuitively, if $w(\mathcal{P}_{\mathcal{H}_t})$ is no larger than the given SPM size, then its colorability is $\mathcal{P}_{\mathcal{H}_t}.freq$. Otherwise, its colorability is approximated as a percentage reduction of $\mathcal{P}_{\mathcal{H}_t}.freq$ in terms of the ratio $\frac{SPM\_SIZE}{w(\mathcal{P}_{\mathcal{H}_t})}$, which represents the percentage of data in $\mathcal{P}_{\mathcal{H}_t}$ that cannot be placed in the SPM.

As a result, the benefit for spilling $v$ from $H_t$ is:

$$v.spillbenefit = (\alpha(\mathcal{H}_t - \{v\}) - \alpha(\mathcal{H}_t)) \times (S_{\text{mem}} - S_{\text{spm}}) \qquad (3)$$

The spilling cost, i.e., penalty incurred by $v$ is estimated by:

$$v.spillcost = v.freq \times (S_{\text{mem}} - S_{\text{spm}}) \qquad (4)$$

where $v.freq$ is the access frequency of array $v$.

We choose a node in $\mathcal{P}_{\mathcal{H}_t}$ to spill such that the spilling profit defined below is maximized among all $v$ in $\mathcal{P}_{\mathcal{H}_t}$:

$$v.spillprofit = v.spillbenefit - v.spillcost \quad (5)$$

A spilled node in $\mathcal{P}_{\mathcal{H}_t}$ is excluded from the current IG $\mathcal{H}_t$. No spilling code needs to be generated. Since a subgraph $\mathcal{G}'$ of a comparability graph $\mathcal{G}$ remains a comparability graph, and a transitive orientation of $\mathcal{G}$ remains transitive in $\mathcal{G}'$ and does not need to be recomputed. However, the heaviest path $\mathcal{P}_{\mathcal{H}_t}$, which may have changed in the current IG, should be recomputed (line 23).

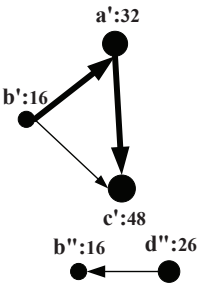



**Figure 10: Interference graph of Figure 6(a) after spilling $d'$.**

For the IG given in Figure 6(a), with $SPM\_Size = 100$, then $w(\mathcal{P}_{\mathcal{H}_t}) = 122 > SPM\_Size$. If spilling is performed according to the heuristic (5), then $d'$ can be selected to spill. We simply remove $d'$ from Figure 6(a), obtaining the resulting graph as shown in Figure 10. The heaviest path now is $b' \rightarrow a' \rightarrow c'$ with a total weight of 96. For the program in Figure 4, no spilling code needs to be generated. We only need to undo the live range splitting for $d$ in loop $\mathcal{L}_1$ and eliminate the hot array $d'$ introduced and the associated copy operations.

## 4.5 Data Tiling with Optimal Tile Sizes

If there are tileable arrays in $TAS_{\mathcal{P}_{\mathcal{H}_t}}$, then data tiling (instead of spilling) can be applied to a selected loop. Data tiling causes some temporary arrays, called *tile arrays*, to be introduced in the selected loop. Their sizes can be expressed as multiples of a tile size variable for the selected loop. The critical path $\mathcal{P}_{\mathcal{H}_t}$ gives an upper bound for the tile size to be used. By combining this upper bound constraint with the cost benefit analysis for all tileable arrays expressed as a function of the tile size variable, the best tile size can be solved analytically or numerically.

The focus of this paper is on demonstrating the feasibility of performing on-demand data tiling together with scratchpad allocation. To this end, we describe a simple approach to selecting which loop to tile and which tileable arrays to tile inside the selected loop. More sophisticated tiling heuristics, as discussed at the end of this section, will be developed in future work.

Let us consider our algorithm in Figure 2 again. If some arrays in $TAS_{\mathcal{P}_{\mathcal{H}_t}}$ can be tiled, we select a loop $\mathcal{L}$ to tile where some tileable arrays in $TAS_{\mathcal{P}_{\mathcal{H}_t}}$ are accessed. Presently, we select $\mathcal{L}$ such that the sum of the access frequencies of all the tileable arrays accessed in $\mathcal{L}$ is the largest. We then undo the live range splitting performed earlier for the tileable arrays accessed in $\mathcal{L}$. Then data tiling, together with loop tiling, is applied to $\mathcal{L}$ with the tile size being symbolically represented by $x$, and the procedure Determine_Optimal_Tile_Size() is called to find the optimal tile size (line 27), as detailed in Figure 11.

The basic idea behind Figure 11 is simple. The weight of a tileable array in $TAS_{\mathcal{P}_{\mathcal{H}_t}}$ is replaced with the weight of its corresponding tiled array. Thus, $w(\mathcal{P}_{\mathcal{H}_t})$ is expressed as a function in terms of the tile size variable $x$. Then a cost-benefit analysis is performed to determine the value of $x$. For every tileable array $\mathcal{A}$ in $\mathcal{L}$, we write $Len(\mathcal{A})$ to represent its original size and $Len(f(\mathcal{A}))$ to represent the size of its corresponding tiled array. Then the access benefit is $\mathcal{A}.freq * (S_{\rm mem} - S_{\rm spm})$ and the transfer cost is $(C_s + C_t * Len(f(\mathcal{A}))) * num\_copies.\mathcal{A} * \frac{Len(\mathcal{A})}{Len(f(\mathcal{A}))}$, where $num\_copies.\mathcal{A}$ denotes the dynamic number of copy operations executed for $\mathcal{A}$ prior to tiling. In line 11, we call Maximize_Profit to find $x$ by maximizing the net profit obtained:

$$\max\left(\sum_{\mathcal{A}\in S_{\rm old}} (\mathcal{A}.freq \times (S_{\rm mem} - S_{\rm spm}) - (C_s + C_t \times Len(f(\mathcal{A}))) \times num\_copies.\mathcal{A} \times \frac{Len(\mathcal{A})}{Len(f(\mathcal{A}))})\right) \quad (6)$$

subject to the following SPM capacity constraint:

$$w(\mathcal{P}_{\mathcal{H}_t}) \leq SPM\_Size$$

and other constraints, including, for example, those constraints related to the sizes of the tileable arrays in $S_{\rm old}$ and the loop bounds.

```
1:  procedure Determine_Optimal_Tile_Size
2:  Input: Loop L to be tiled and the heaviest path P_{H_t}
3:  Output: Optimal tile size x
4:  Let S_old be the set of m tileable arrays in L of TAS_{P_{H_t}}
5:  Let S_new be the set of m new "tiled arrays"
6:  Let f be a bijective function from S_old to S_new
7:  for every array A ∈ S_old do
8:      Replace the weight of A with the size of f(A)
9:  end for
10: Recalculate w(P_{H_t}) (which is now a function of x)
11: x = Maximize_Profit()
12: return x
13: end procedure
```

**Figure 11: Optimal tile size selection.**

Let us consider the IG in Figure 6(a) with the heaviest path being $\mathcal{P}_{\mathcal{H}_t} = b' \rightarrow d' \rightarrow a' \rightarrow c'$ and $w(\mathcal{P}_{\mathcal{H}_t}) = 122$. Let the SPM size be 64. Some arrays must be tiled to be placed in the SPM. Suppose that $\mathcal{L}_1$ is selected with $S_{\rm old} = \{a', b', c', d'\}$. Let $S_{\rm new} = \{ta', tb', tc', td'\}$ be the set of the corresponding tiled arrays introduced. Applying data tiling, together with loop tiling, to $\mathcal{L}_1$ yields the program in Figure 12. As a result, the IG shown in Figure 6(a) evolves into the one shown in Figure 13, giving rise to $w(\mathcal{P}_{\mathcal{H}_t}) = 7x$.

When performing the cost benefit analysis, the transfer cost is $8640/x + 5760$ and the access benefit is $16 * 4 * 5 * (100 - 1) = 31680$. So the net profit to be maximized is $31680 - 8640/x - 5760 = 25920 - 8640/x$ subject to the space constraint $7x \leq 64$. To be practical, the tile size $x$ is required to be at least 2. So the optimal tile size for $\mathcal{L}_1$ is $x = 8$ as shown in Figure 14.

Let us return to our algorithm in Figure 2. Once data tiling has been performed, we go back in line 28 to line 2 to rebuild the IG and repeat the same process until the weight of the heaviest path in the current IG is no longer larger than the SPM size.

Our simplistic tiling heuristics can be further improved along a number of directions. The search space for data tiling is defined by a number of factors, including, which loop(s) to tile, which tileable

```
1    float a[32], b[16], c[48], d[26];
2    int i, sum;
3    ...
4    float ta'[2x], tb'[x], tc'[3x], td'[x];
5    for (i = 1:16:x) {
6       load_tile(a[i:i+2x-1], ta');
7       load_tile(b[i:i+x-1], tb');
8       load_tile(c[i:i+3x-1], tc');
9       load_tile(d[i:i+x-1], td');
10      for (j = 1:x) {
11         tb'[j] = tb'[j] + ta'[2j]*tc'[3j] + td'[j];
12      }
13      store_tile(tb', b[i:i+x-1]);
14      store_tile(td', d[i:i+x-1]);
15   }
16   ...
```

**Figure 12: The program after data tiling (applied to $\mathcal{L}_1$).**

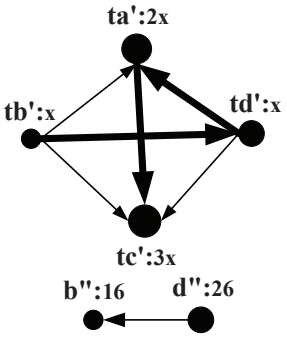



**Figure 13: The heaviest path after data tiling (applied to $\mathcal{L}_1$).**

array(s) in a selected loop to tile, what tile size to use to tile a selected loop, and what tile size, i.e., array size to use for a selected tileable array in a selected loop. The most aggressive option is to analytically try all possibilities and pick the best according to some heuristics employed. The most computationally-efficient option could be to randomly pick one loop and one contained tileable array in the loop to tile. The solution introduced above represents a simple compromise. Better tiling heuristics that are useful for on-demand data tiling will be developed in future work.

# 5. EXPERIMENTS

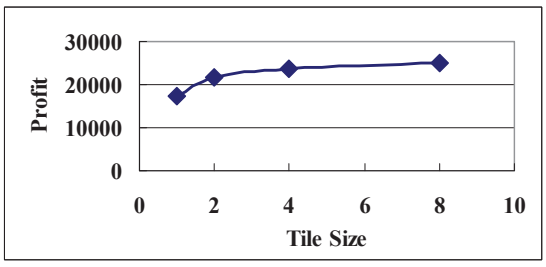



**Figure 14: The net profits calculated according to (6) for different tile sizes.**

We have modified SimpleScalar to integrate SPM instead of cache. There are four parameters to be considered. The cost of communicating $n$ bytes between the SPM and off-chip memory is estimated as $C_s + C_t \times n$ in cycles. Two other parameters are $S_{spm}$ and $S_{mem}$, 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 set to be $C_s = 90, C_t = 10, S_{spm} = 1$ and $S_{mem} = 100$.

We have implemented our algorithm in the SUIF compiler infrastructure. We take a C program as input, perform a source-to-source transformation by applying our algorithm, and finally, produce as output a new C program with SPM operations inserted. Then the program is compiled by GCC and run on SimpleScalar. If the program is originally in FORTRAN, then the f2c tool is used to convert it to C code first.

The benchmarks, which are all taken from SPEC2000, are the computation-intensive procedures of the corresponding benchmarks. In more detail, *resid* and *psinv* are from *172.mgrid*, *buts* from *173.applu*, *calc1* and *calc2* from *171.swim*, *zaxpy* is from *168.wupwise*.

These benchmarks are modified with all arrays of more than one dimension being replaced by 1-D arrays so that data tiling can be applied by our algorithm.

## 5.1 Performance

Figure 15 gives the performance speedups over the non-tiling graph coloring SPM allocation algorithm [13]. For *resid*, when the SPM size is smaller than 256KB without tiling, none or only some of the arrays can be placed in SPM. Therefore, our algorithm achieves relatively large speedups in these cases. When the SPM is 256KB or larger, all the arrays can be placed in SPM. So both algorithms achieve the same performance. For *psinv*, the same trend is observed except that the demarcation line is 128KB. We have identified *buts* as an interesting application. The computation is performed in a five-level loop nest. The three innermost loops access consecutive array elements. However, the two outermost loops access array elements with a large non-unity stride. According to the cost-benefit analysis, it is not worth it to copy all the arrays accessed in the loop nest into SPM. However, it is beneficial to copy the array elements accessed in the three innermost loops into SPM (in which case data tiling is needed), requiring only a small SPM (smaller than 16KB). In addition to this loop nest, there are some other small loop nests containing arrays that can all fit into a 16KB SPM. That is why under different SPM sizes, our algorithm achieves the same speedup. For *calc1*, without tiling, no array can be placed in the SPM even with a size up to 512KB, in which case, the performance is equal to the one when the SPM is not used. When data tiling is used, the situation is different. As the SPM size increases, the optimal tile size becomes larger and larger, and consequently, the number of copy operations becomes smaller and smaller, as validated in Figure 16. That is why our algorithm achieves increasingly better speedups when the SPM size increases. Finally, *calc2* and *zaxpy* are similar to *calc1*, except that *zaxpy* demonstrates a much lower computation intensity.

Figure 17 gives the execution times when the SPM sizes range from 16KB to 512KB, normalized to the execution time obtained with an infinite large SPM. By demonstrating performance close to the best possible, we show that our algorithm makes a good utilization of the limited SPM space.

## 5.2 The Impact of Live Range Coalescing

Figure 18 demonstrates the impact of live range coalescing on performance for *resid* through eliminating unnecessary copy operations. As described before, when the SPM size reaches 128KB

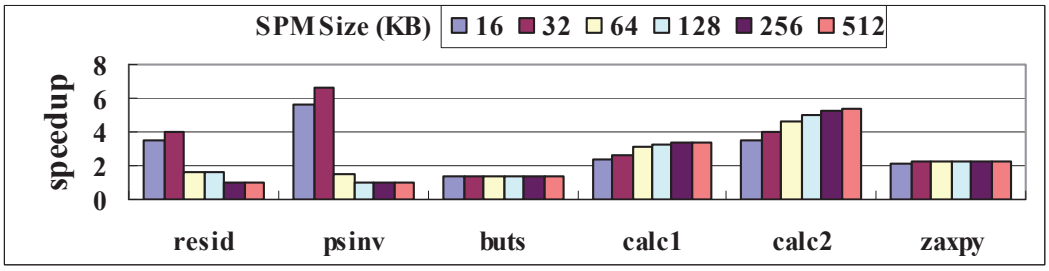



**Figure 15: Performance speedups over the non-tiling graph coloring technique [13].**

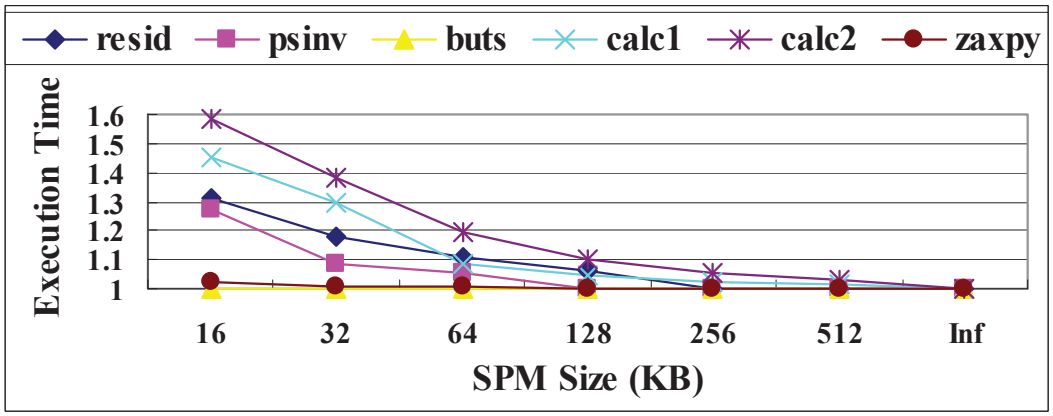



**Figure 17: Execution times normalized to the execution time obtained when the SPM size is $+\infty$.**

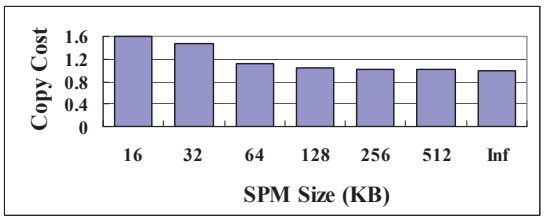



**Figure 16: Copy cost under different SPM sizes for *calc1*.**

or beyond, some or all of the arrays can be placed in the SPM. In addition, their live ranges may be selectively coalesced with the corresponding arrays in move-related nodes.

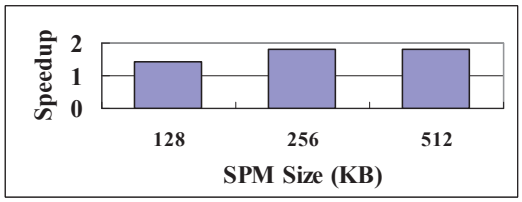



**Figure 18: Speedups for *resid* calculated by execution time (uncoalesced) / execution time (coalesced).**

## 6. CONCLUSION

This paper proposes a new comparability graph coloring allocator that integrates data tiling and SPM allocation for arrays by tiling arrays on-demand to improve utilization of the SPM. The novelty lies in repeatedly identifying the heaviest path in an array interference graph and then reducing its weight by tiling certain arrays on the path appropriately with respect to the size of the SPM. The effectiveness of our algorithm is validated by using a number of selected benchmarks for which existing algorithms are ineffective.

## 7. ACKNOWLEDGMENTS

This research is supported in part by the Funds for Creative Research Groups of China (60921062), the National Natural Science Foundation of China (60873014), and an Australian Research Council Grant (DP0881330).

## 8. REFERENCES

[1] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002.

[2] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[3] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982*

*SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[4] J.d. Cuvillo, W. Zhu, H.u. Ziang, and G.R. Gao. Fast: A functionally accurate simulation toolset for the cyclops64 cellular architecture. In *MoBS2005: Workshop on Modeling, Benchmarking, and Simulation*, pages 11–20. ACM Press, 2005.

[5] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, and Jung-Ho Ahn et al. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 35–42, 2003.

[6] Janet Fabri. Automatic storage optimization. *SIGPLAN Not.*, 14(8):83–91, 1979.

[7] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., 2004.

[8] S. Louis Hakimi, Edward F. Schmeichel, and Neal E. Young. Orienting graphs to optimize reachability. *Inf. Process. Lett.*, 63(5):229–235, 1997.

[9] Pinar Heggernes, Federico Mancini, and Charis Papadopoulos. Minimal comparability completions of arbitrary graphs. *Discrete Appl. Math.*, 156(5):705–718, 2008.

[10] Qingguang Huang, Jingling Xue, and Xavier Vera. Code tiling for improving the cache performance of pde solvers. In *ICPP*, pages 615–624, 2003.

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

[12] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. *SIGPLAN Not.*, 32(5):346–357, 1997.

[13] Lian Li, Hui Feng, and Jingling Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim.*, 6(3):1–17, 2009.

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

[15] Lian Li, Quan Hoang Nguyen, and Jingling Xue. Scratchpad allocation for data aggregates in superperfect graphs. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 207–216. ACM, 2007.

[16] Lian Li, Hui Wu, Hui Feng, and Jingling Xue. Towards data tiling for whole programs in scratchpad memory allocation. In *ACSAC'07: Proceedings of the 12th Asia-Pacific Computer Systems Architecture Conference*, pages 63 – 74, 2007.

[17] J. Makino, K. Hiraki, and M. Inaba. Grape-dr: 2-pflops massively-parallel computer with 512-core, 512-gflops processor chips for scientific computing. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–11. ACM, 2007.

[18] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[19] Sumesh Udayakumaran and Rajeev 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, New York, NY, USA, 2003. ACM.

[20] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[21] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109. ACM, 2004.

[22] Xuejun Yang, Li Wang, Jingling Xue, Yu Deng, and Ying Zhang. Comparability graph coloring for optimizing utilization of stream register files in stream processors. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 111–120, 2009.

[23] Chunhui Zhang and Fadi Kurdahi. On combining iteration space tiling with data space tiling for scratch-pad memory systems. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 973–976. ACM, 2005.