MGC: Multiple Graph-Coloring for Non-Volatile Memory Based Hybrid Scratchpad Memory *

Qingan Li[1,2], Yingchao Zhao[1], Jingtong Hu[3], Chun Jason Xue[1], Edwin Sha[3,4],Yanxiang He[2]
[1]Department of Computer Science, City University of Hong Kong, Hong Kong
[2]School of Computer Science, Wuhan University, China
[3]Department of Computer Science, University of Texas at Dallas, USA
[4]College of Computer Science, Chongqing University, China
{lqingan2, yingzhao, jasonxue}@cityu.edu.hk, {jthu, edsha}@utdallas.edu, yxhe@whu.edu.cn

Abstract

Scratchpad Memory (SPM), a software-controlled on-chip memory, has been widely used as an alternative to caches in modern embedded systems due to its energy efficiency. To further reduce the energy consumption, non-volatile memory (NVM) based hybrid SPM has been proposed recently. This paper targets the problem of allocating program variables into hybrid SPM based systems. Both an ILP formulation and a graph-coloring based algorithm are proposed. The experiments show that the proposed graph-coloring framework achieves both better memory latency and lower energy costs in comparison to previous works.

1 Introduction

Energy consumption is a crucial issue in the design of embedded systems. On-chip caches typically consume 25% to 45% of the total chip power [15]. Scratchpad Memory (SPM), a software-controlled on-chip memory, is often applied for energy efficiency in embedded systems. The work by Banakar et al. [4] demonstrates that SPMs are more efficient than caches in performance, energy consumption, and area costs. Software techniques for SPMs also guarantee better timing predictability than a cache memory, which is critical in hard real-time systems. Given these advantages, SPMs are used as alternatives to caches in modern embedded processors such as Motorola M-core MMC221 and TI TMS370Cx7x. This paper proposes techniques for allocating program variables on hybrid SPMs.

*This work is partially supported by the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 123609, CityU 123210, CityU 123811], and the National Natural Science Fund of China [Project No. 61170022, 61173014, 61133005], and NSF CNS-1015802, Texas NHARP 009741-0020-2009.

Traditional SPMs are fabricated in pure SRAM. As the number of CMOS transistors increases, leakage power consumption becomes a critical issue. Non-volatile memories (NVMs), with the features of low leakage power and high-density, present a new way of addressing the memory leakage power consumption problem. However, the write operations on NVM suffer from considerably higher energy and longer latency. Recently, hybrid SPMs consisting of SRAM and NVM is proposed [8] and smart memory management techniques are needed when NVM is applied.

Much work has been done for SPM allocation. Avissar et al. derive a static allocation approach, in which the allocation of data to SPM is fixed and unchanged throughout program execution [3]. With the assumption that the live range of each data object is through the whole program execution, they model the problem as an Integer Linear Programming (ILP) problem, and solve it using commercial packages. Considering dynamic program behavior, Udayakumaran et al. present a dynamic allocation method for global and stack data [17]. Hu et al. propose a novel hybrid SPM architecture consisting of NVM and SRAM, and present a dynamic allocation approach, where the code is partitioned into multiple regions, and for each region a dynamic programming algorithm is used to reallocate the data [8]. Their solution assumes that the live range of each data objects is through the whole program execution, so cannot explore the opportunity that data objects with disjoint live ranges can share the same memory addresses in SPM. The work by Li et al. considers live ranges of variables, and can allocate arrays with non-overlapped live ranges into the same memory address [13]. There is an implicit assumption in their solution that, the cost of access to one memory unit is always better than another memory unit. This assumption is reasonable for the problem of register allocation, where register is always superior to main memory, and for pure SRAM based SPM, where SPM is always superior to off-

chip memory. However, this assumption does not always hold in a hybrid SPM based system. For example, in a hybrid SPM consisting of NVM and SRAM, the latency of a write operation to NVM is not always cheaper than off-chip memory. Furthermore, when the cost considered is parameterized as a combination of latency, energy, area, etc., it is uncommon that one memory is always better than the other. Therefore, new management schemes are needed for hybrid SPM.

In this paper, we propose a compilation technique for statically allocating program data into a hybrid SPM to minimize the total memory cost. The memory cost can be latency, power consumption, or any combination. In this paper, we model the allocation problem using an ILP formulation. Unlike the ILP solution presented in [3], the ILP formulation proposed in this paper take into account of live ranges of data objects. Furthermore, as ILP does not scale well with large programs, we also propose a polynomial-time heuristic algorithm, *Multiple Graph-Coloring* (MGC), to allocate program variables in hybrid SPMs. In MGC, the interference graph of variables is split into vertex-induced subgraphs according to each variable's favorite memory unit, and then a graph-coloring based algorithm is applied for each subgraph. Memory cost is integrated into this graph-coloring process. Experimental results show that MGC achieves better execution latency and lower energy costs in comparison to previous works.

The major contributions of this paper include:

1. For the first time, an ILP formulation exploring that variables that are not alive simultaneously can share the same memory address, is proposed for the problem of hybrid SPM allocation.

2. A graph-coloring based polynomial-time algorithm is proposed for the problem of hybrid SPM allocation.

The rest of this paper is organized as follows. Problem formulations and backgrounds are introduced in Section 2. The ILP formulation and the MGC algorithm are presented in Section 3 and Section 4 respectively. Section 5 shows the experiments. Finally, Section 7 concludes this paper.

2 Problem Formulations and Backgrounds

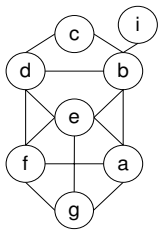
Given a program, and a hybrid SPM, a compilation approach is proposed in this paper to allocate program data into hybrid SPM with objective of minimizing the cost of memory accesses (the cost could be latency, power consumption or any combination). There are two assumptions for this work. First, both local data and global data need static allocation. This assumption is reasonable for embedded systems, since many low end microcontrollers (MCUs), such as the 8-bit PIC MCUs [1], do not support stack storage for local data. Second, the architecture parameters are known at the compilation time. The architecture parameters include the types of memory units used (SRAM, NVM, etc.), and the size as well as the cost for accessing each memory unit. Note that there is no assumption that one memory unit always dominates the other, which makes this problem is different from register allocation as well as pure SRAM SPM allocation. Therefore, the solutions for this problem requires more than a simple cascading application of the classic graph coloring algorithm.

To exploit the live ranges of variables, live analysis is used. Live analysis is a classic data flow analysis performed by compilers to calculate program points where each variable is alive. Each variable lives through a set of program points, which constitute the live range of that variable. If the intersection of two variables's live ranges is not empty, these two variables interfere with each other. This interference relation could be represented by an interference graph, where each node represents a variable in the program, and the undirected edges connect pairs of nodes which interfere. Fig. 2(a) shows the interference graph for the example code in Fig. 1.

To minimize the total cost of a memory system, it is beneficial to know the access frequency of each variable during program execution. Static profiling [18], which estimates the execution frequency at compilation time, is employed in this paper to obtain such information. Fig. 2(b) shows the access frequency for the code in Fig. 1.

```
 1  b = 4;
 2  i = 0;
 3  while(i < 15 )
 4  {
 5     i = i + 1;
 6     array[i] = b;
 7  }
 8  a = 3;
 9  c = a * b + a + b;
10  d = b * c;
11  c = b + d;
12  e = c * d + d/2;
13  f = b * d + 2 * e;
14  g = d + 5 * e;
15  a = 2 + g;
16  return a * f + e * g;
```

Figure 1. An example program.



(a) Interference graph.

Data	Read	Write
a	3	2
b	20	1
c	2	2
d	5	1
e	3	1
f	1	1
g	2	1
i	31	16

(b) Number of accesses.

Figure 2. Interference graph and access frequency.

3 The ILP Formulation

In this section, we present the ILP formulation for the problem of allocating program data in a hybrid SPM system. There are two kinds of constraints for this problem. The *allocation constraints* ensure that each variable can be allocated into only one memory unit. The *interference constraints* ensure that variables alive simultaneously cannot share the same memory address. The following notations are used, and are assumed constant in the ILP formulation.

- n: number of variables
- m: number of memories
- $U = \{u_1, \ldots, u_m\}$: set of memories
- P_i: size of u_i in byte
- CR_i: cost of a read from u_i
- CW_i: cost of a write to u_i
- $V = \{v_1, \ldots, v_n\}$: set of variables
- Q_i: size of v_i in byte
- NR_i: number of times v_i is read
- NW_i: number of times v_i is written
- $\mathit{Inter} = \{(v_i, v_j) | \textit{if } v_i \textit{ interferes with } v_j\}$: pairs of interfered variables

Equation 1 is used to calculate the cost for a variable v_i in a memory unit u_j.

$$\mathit{MemoryCost}_{i,j} = (NR_i \cdot CR_j + NW_i \cdot CW_j) \cdot Q_i \quad (1)$$

We use $a_{i,j}$ to describe whether variable v_i is assigned to memory unit u_j, as defined in Equation 2.

$$a_{i,j} = \begin{cases} 1 & \text{if variable } v_i \text{ is assigned to memory } u_j. \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Then, the *allocation constraints* is formulated as Equation 3.

$$\sum_{j=1}^{m} a_{i,j} = 1 \quad \forall i \in \{1, 2, \ldots, n\} \quad (3)$$

We use s_i to denote the start address of the memory space allocated for variable v_i. If variable v_i is allocated to memory u_j, then we can see that s_i must be in the range $[0, P_j - Q_i]$. We can use the Inequation 4 to describe this constraint. Here, B is a large enough number, and we can simply take $B = \sum_{i=1}^{m} P_i$. If $a_{i,j} = 1$, which means variable v_i is assigned to memory u_j, then Inequation 4 becomes $0 \leq s_i \leq P_j - Q_i$.

$$\begin{cases} s_i \geq (a_{i,j} - 1) \cdot B & \forall j \in \{1, 2, \ldots, m\} \\ s_i \leq P_j - Q_i + (1 - a_{i,j}) \cdot B & \forall j \in \{1, 2, \ldots, m\} \end{cases} \quad (4)$$

For each pair of interfered variables v_i and v_j, if they are allocated in the same memory, their starting addresses s_i and s_j must satisfy either $s_i + Q_i \leq s_j$ or $s_j + Q_j \leq s_i$. To indicate the order of such two variables, we define $y_{i,j}$ as Equation 5.

$$y_{i,j} = \begin{cases} 1 & \text{if } s_i < s_j. \\ 0 & \text{if } s_i > s_j \end{cases} \quad (5)$$

Then, the *interference constraints* for any pair of interfered variables (v_i, v_j) can be formulated as Inequation 6. Notice that Inequation 6 must hold for any memory u_t. If two variables are allocated in different memories, i.e., $a_{it} + a_{jt} < 2$, then Inequation 6 is naturally held. If they are allocated into the same memory, then Inequation 6 leads to $s_i + Q_i \leq s_j$ when $y_{i,j} = 1$, and $s_j + l_j \leq s_i$ when $y_{i,j} = 0$. Hence Inequation 6 guarantees that interfered variables cannot share the same memory address.

$$\begin{cases} s_i + Q_i \leq s_j + (1 - y_{i,j}) \cdot B + (2 - a_{i,t} - a_{j,t}) \cdot B \\ s_j + Q_j \leq s_i + y_{i,j} \cdot B + (2 - a_{i,t} - a_{j,t}) \cdot B \end{cases} \quad (6)$$

The objective is to minimize the total cost of all memory access, which could be formulated as below:

$$\sum_{i=1}^{n} \sum_{j=1}^{m} a_{i,j} \cdot \mathit{MemoryCost}_{i,j} \quad (7)$$

4 MGC: Multiple Graph-Coloring Algorithm

Similar to register allocation, SPM allocation can also be viewed as a graph coloring problem. The interference graph of program variables is the graph to be colored. Each type of memory is associated with a set of colors, and the number of colors in this set equals the size of this memory. In the context of hybrid SPM, each coloring has a cost, which corresponds to the memory cost. Coloring a node with colors

from the same set results in the same cost, as they represent allocations into the same memory unit. Therefore, the hybrid SPM allocation problem could be restated as: Given the interference graph, and the sets of colors, how to color this graph such that nodes connected by the same edge could not be colored with the same color, while the total cost is minimized?

It is observed that a good allocation has two features. First, it tries to allocate each variable into its preferred memory unit. Second, when two variables, v_i and v_j, compete for a low-cost memory unit u_c, if the cost penalty resulting from spilling v_i is more than from spilling v_j, then, v_i carries a preference to be allocated into u_c. In this section, based on the graph-coloring framework, a polynomial-time heuristic algorithm, Multiple Graph-Coloring (MGC), is proposed for the hybrid SPM allocation. The input to MGC algorithm includes: the access frequency for each variable, the interference graph for variables, and the architecture parameters. As stated in Section 2, the information about access frequency can be obtained from static profiling [18], and the interference graph is constructed by live analysis. The MGC algorithm, as detailed in Algorithm 4.1, mainly consists of two steps: computing and sorting the memory cost for each variable, and graph-coloring. The graph-coloring process splits the graph into subgraphs, and then coloring and spilling is done for each subgraph. In the rest of this section, these two main steps are discussed first, and then a walk-through example is presented.

4.1 Computing and Sorting the Access Cost

Given the access frequency, the memory cost for each variable could be calculated using Equation 1. Then, each variable v_i is associated with a memory cost list, $\{$*MemoryCost*$_{i,1}$, *MemoryCost*$_{i,2}$, ..., *MemoryCost*$_{i,k}\}$, where k is the number of memory units. In addition, the memory cost list for each variable is sorted in ascending order. *MemoryCost*$_{i,m}$ precedes *MemoryCost*$_{i,n}$ in the memory cost list associated with v_i, if and only if *MemoryCost*$_{i,m}$ is less than *MemoryCost*$_{i,n}$.

4.2 Graph-Coloring Process

The graph-coloring process is shown in Algorithm 4.2. It consists of a loop which includes two parts: the first part is to split the interference graph into subgraphs, and the second part is to color and spill the nodes of each subgraph respectively. The overall graph-coloring process for allocation of a hybrid SPM consisting of SRAM and NVM is shown in Fig. 3.

Subgraph splitting. Each variable has its preferred memory unit. According to this preference, the interference graph could be split into vertex-induced subgraphs, where the variables preferring the same memory unit are split into the same subgraph. Each subgraph corresponds to a memory unit.

Coloring and spilling for each subgraph. This process is detailed in Algorithm 4.2. It consists of four stages: simplify, potential spill, select, and actual spill. The naming as well as the meanings of these four stages is similar

Algorithm 4.1 The MGC Algorithm.

Input:
MemoryType: memory units
MemoryCost: pair of (*MemoryType*, **double**)
MemoryCostList: map from nodes to list of *MemoryCost*
InterGraph: list of nodes (using a adjacency list to represented the annotated interference graph)
SubGraphList: map from *MemoryType* to list of nodes (each list represents a subgraph)

Output:
DataAlloc: map from nodes to colors

```
1:  // Step 1: computing and sorting the access cost
2:  compute access cost for each variable using Eq. 1;
3:  sort memory cost list for each variable in ascend order;
4:  // Step 2: multiple graph-coloring
5:  bool bActualSpilled ← true;
6:  repeat
7:     bActualSpilled ← false;
8:     initialize DataAlloc to be empty;
9:     // Part I: splitting the interference graph
10:    split the InterGraph into SubGraphList;
11:    if SubGraphList is empty then
12:       print "Out of memory error";
13:       return false;
14:    end if
15:    // Part II: Coloring and spilling for each subgraph
16:    for each (memoryType, subgraph) of SubGraphList do
17:       // ColorSpill is illustrated in Algorithm 4.2
18:       if ColorSpill(memoryType, subgraph) then
19:          bActualSpilled ← true;
20:       end if
21:    end for
22: until bActualSpilled is false;
23: return true;
```

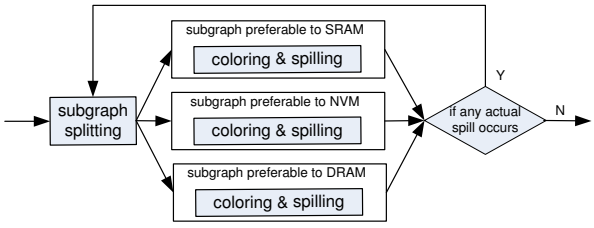


Figure 3. Graph-coloring for a hybrid SPM consist of SRAM and NVM.

to that in graph-coloring algorithm for register allocation. For each subgraph, the simplify process finds nodes with degree less than k, where k is the size of the memory unit related to this subgraph. When such a node is found, it is removed from the subgraph and pushed into a stack, and all of the edges connected to it are removed too. This process is repeated until none of the remaining nodes could be simplified. If there are nodes which cannot be simplified, one of them is marked as potential spill node, removed from the subgraph and pushed into the stack. This process is repeated until there exist nodes with degree less than k, at which point we return to the simplify stage. We choose the potential spill node by calculating the spill cost of each node using Equation 8. Here, the penalty is the difference of cost corresponding to a node's first and second favorite memory units. The node with the lowest spill cost will be chosen as the potential spill node.

$$spill\ cost = penalty/degree \tag{8}$$

After the simplify and potential spill stages, all nodes would be removed from the subgraph. We then assign colors to nodes in the order popping them off the stack, under the constraint that nodes connected by the same edge cannot be colored with the same color. When a node cannot be colored, it is marked as an actual spill node, and left uncolored. Note that an actual spill node can not be allocated into its currently favorite memory unit, so the actual spill here causes the first element from the memory cost list of this node to be erased. If any actual spill occurs in a subgraph, there is a need to return to the subgraph spitting phase as illustrated in Algorithm 4.1.

Algorithm 4.2 ColorSpill.

Input:
SubGraph: list of nodes (using a adjacency list to represented the subgraph)
MemoryType: memory unit associated with this subgraph
MemoryCost: pair of (*MemoryType*, **double**)
MemoryCostList: map from nodes to lists of *MemoryCost*
Stack: a stack for temporarily storage of the nodes
Output:
SubGraph: list of nodes with updated memory cost list

```
1: // Simplify and potential spill
2: while SubGraph is not empty do
3:    if there is a node n_i with degree less than MemoryType.size then
4:       push n_i into Stack;
5:       remove n_i from SubGraph;
6:    else
7:       choose a potential spill node n_p using Equation. 8;
8:       push n_p into Stack;
9:    end if
10: end while
11: // Select and actual spill
12: bool bActualSpill ← false;
13: while Stack is not empty do
14:    pop out one node n_i from Stack;
15:    if there is a color c for n_i then
16:       DataAlloc[n_i] ← c;
17:    else
18:       bActualSpill ← true;
19:       remove the first element from MemoryCostList[n_i];
20:    end if
21: end while
22: return bActualSpill;
```

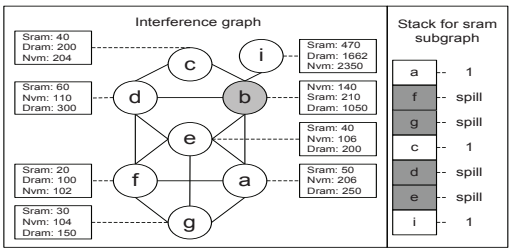
4.3 A Walk-Through Example

Table 1. Architecture parameters for the example.

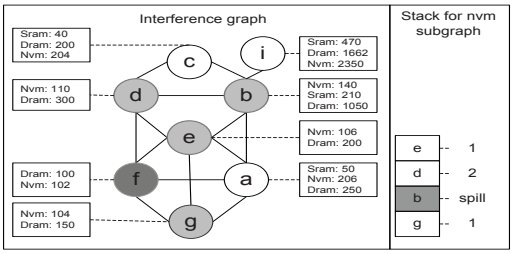
Memory	Read cost	Write cost	Size (byte)
on-chip SRAM	10	10	1
on-chip NVM	2	100	2
off-chip DRAM	50	50	4

A walk-through example is presented in this subsection. The access frequency and interference graph are given in Fig. 2. The architecture parameters assumed are shown in TABLE 1. Memory cost of each variable for each memory is calculated using Equation 1, as illustrated in Fig. 4(a).

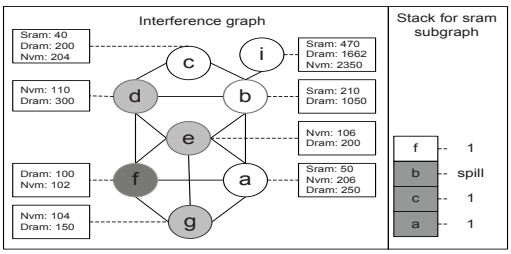
First, the interference graph is split into subgraphs according to each node's favorite memory. In Fig. 4(a), only node b prefers NVM, so it is moved into the NVM subgraph. All the other nodes are moved into the SRAM subgraph. The nodes preferring the same memory are highlighted in the same color. Then, the coloring and spilling process continues for each subgraph. For the NVM subgraph, one color from the color set related to NVM needs to be assigned to b. For the SRAM subgraph, since the size of SRAM is one byte, k equals 1. It is found that the degree of node i is less than k. So, i is removed from this subgraph and pushed into the stack. Then, we need to choose a potential spill node. Among all the remaining nodes, the spill cost of node e is the lowest, since the penalty is 66 (106-40), the degree is 4, and thus the spill cost is 16.5 (66/4). Then node e is chosen as a potential spill node and removed from this subgraph. Afterwards, there is no node with degree less than k, so we continue to mark the next potential spill node. This process continues by potentially spilling d, simplifying c, potentially spilling g and f, and finally simplifying a. The complete process is shown in Fig. 4(a), where the potential spilled nodes are marked gray in the stack. After all nodes have been removed, we try to assign colors to the nodes in the order popping them off the stack. It is found all the potentially spilled nodes need to be actually spilled,



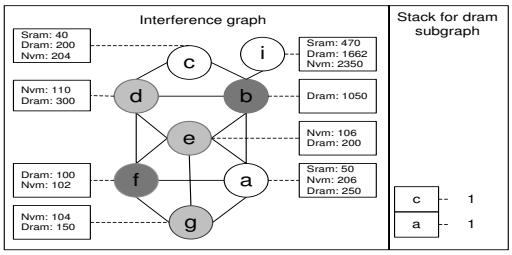
(a) The first iteration of MGC.



(b) The second iteration of MGC.



(c) The third iteration of MGC.



(d) The fourth iteration of MGC.

Figure 4. A walk-through example.

Sram (size=1)	{a,c,i}			
Nvm (size=2)	{e,g}	{d}		
Dram (size=4)	{b,f}			

Figure 5. Final allocation of the example.

and the coloring is shown in Fig. 4(a). Now, $\{a, c, i\}$ is allocated into SRAM with the same address, $\{b\}$ is allocated into NVM. The updated interference graph is shown in Fig. 4(b). Since there are actual spills, we return to the subgraph splitting phase and start a new iteration. For the new subgraph splitting phase, a, c, i prefer SRAM; b, d, e, g prefer NVM; and f prefers DRAM. Then the coloring and spilling process for each subgraph continues. In this example, four iterations are needed in total, as illustrated in Fig. 4. The final allocation is shown in Fig. 5. It is found that only four bytes are used to hold eight variables, and only two variables.

5 Experiments

The experimental platform is based on LLVM compiler infrastructure [10]. A static profiling algorithm based on [18] has been implemented and is used to estimate the access frequency of each variable. Live analysis is also implemented to obtain the interference graph. In this paper's implementation, only local scalar variables, including both source-language variables and compiler-generated temporaries, are considered. But, the proposed method will also work for global variables.

Table 2. Architecture parameters in experiments.

Memory Type	Read/Write latency (ns)	Read/Write energy (nJ)
on-chip sram	3.95/3.95	0.034/0.034
on-chip pcm[1]	1.55/{131.01, 61.01}	0.043/{3.21, 3.85}
main memory	104.4/104.4	3.26/3.26

In the experiments, a hybrid SPM which consists of SRAM and PCM [8] is evaluated. The PCM memory simulator *NVsim* [5], which is a PCM-supporting variant of the CACTI tool, is used to estimate the read/write latencies and the energy consumption for a PCM memory of given size. 45 *nm* technology is used with the tool. NVsim is also used to obtain the access latencies and energy consumption for a given size of SRAM memory. The parameters obtained are shown in TABLE 2.

Five SPM allocators are implemented to evaluate the proposed work. The *MGC allocator* implements the MGC algorithm presented in Section 4. The *ILP allocator* implements the ILP formulation presented in Section 3. This allocator works as a baseline to evaluate the other allocators. The *ILP-N allocator* implements the algorithm presented in [3], which assumes that the live range of each program data object is through the whole execution time of the pro-

[1]The write cost for PCM includes two parts, the first is for setting one, and the second is for resetting to zero.

Table 3. Memory latency normalized to the ILP allocator.

benchmark	ILP-N(%)		ODA(%)		MC(%)		MGC(%)		MGC vs MC(%)	
	config1	config2	config1	config2	config1	config2	config1	config2	config1	config2
basicmath	905.3	505.3	666.7	296.4	101.0	100.9	101.9	100.0	**100.9**	**99.1**
susan	304.8	154.3	–	–	109.1	109.1	100.0	100.0	**91.7**	**91.7**
dijkstra	884.8	425.9	877.9	419.1	104.8	100.1	101.5	100.0	**96.8**	**99.9**
stringsearch	780.4	380.2	780.4	380.3	108.4	104.9	101.8	100.0	**93.9**	**95.3**
sha	509.7	347.2	317.0	197.6	120.3	101.6	108.0	100.4	**89.8**	**98.8**
CRC32	555.1	679.8	573.7	743.0	140.4	153.3	100.1	100.0	**71.3**	**65.2**
FFT	1336.0	687.5	1053.0	484.8	130.5	103.9	114.1	100.0	**87.4**	**96.2**
adpcm	575.3	795.8	553.5	763.6	122.4	117.6	127.9	117.5	**104.5**	**100.0**
Average(%)	731.4	497.0	688.9	469.3	117.1	111.4	**106.9**	**102.3**	**92.0**	**93.3**

Table 4. Memory dynamic energy consumption normalized to the ILP allocator.

benchmark	ILP-N(%)		ODA(%)		MC(%)		MGC(%)		MGC vs MC(%)	
	config1	config2	config1	config2	config1	config2	config1	config2	config1	config2
basicmath	3239.2	1859.0	2454.6	758.2	105.6	100.0	105.6	100.0	**100.0**	**100.0**
susan	899.8	316.4	–	–	114.5	114.5	100.0	100.0	**87.3**	**87.3**
dijkstra	3365.8	1575.0	3347.5	1556.5	121.7	100.0	106.9	100.1	**87.8**	**100.1**
stringsearch	2869.9	1341.2	2876.2	1345.8	128.5	100.0	106.1	100.0	**82.5**	**100.0**
sha	1670.8	1107.8	988.0	526.2	191.5	106.0	138.6	102.3	**72.4**	**96.5**
CRC32	659.5	1177.0	682.7	1283.2	155.5	206.9	153.6	109.9	**98.7**	**53.1**
FFT	4744.4	2560.3	3805.8	1583.1	232.1	100.8	175.5	100.1	**75.6**	**99.4**
adpcm	622.2	1153.2	599.6	1108.9	128.7	128.8	117.2	116.2	**91.1**	**90.2**
Average(%)	2259.0	1386.2	2107.8	1166.0	147.3	119.6	**125.4**	**103.6**	**86.9**	**90.8**

gram. Both ILP based allocators use Lingo [2] to obtain the results. The *ODA allocator* implements the algorithm presented in [8]. The *MC allocator* implements the Memory Coloring (MC) algorithm presented in [13]. As discussed in Section 1, the memory coloring algorithm statically assumes the preference for memory units. In the experiments, the MC allocator assumes SRAM is preferable to PCM, and PCM is preferable to off-chip memory. All of the five allocators output the allocation information which associates each variable with a memory address. A simulator is designed to accept this information as the input, and evaluate the total memory cost using Equation 7. All the benchmarks used are from MiBench [6].

To evaluate the memory latency and dynamic energy consumption, two sets of experiments are conducted respectively. We design two memory configurations for both sets of experiments. In the first configuration, the size of SRAM space is 5% and the size of NVM space is 10% of the total data size for each benchmark. In the second configuration, the size of SRAM space is 10% and the size of NVM space is 20% of the total data size for each benchmark. The results are shown in TABLE 3 and TABLE 4. The benchmarks used are shown in the first column. Then, normalized to the *ILP allocator*, the cost of the other four allocators are listed in the following columns, respectively. The improvement of the *MGC allocator* over the *MC allocator* is shown in the last two columns.

It is found that the *ILP-N allocator* and the *ODA allocator* do not perform adequately well for the selected benchmarks. There are two reasons. First, in embedded systems, the register resources are very restricted, and thus in the experiments no register allocation is conducted. Therefore, there is still a large number of compilation temporaries even with traditional optimization techniques enabled. Second, both the *ILP-N allocator* and the *ODA allocator* do not consider the live ranges of variables, thus resulting of much higher memory pressure.

Considering memory latency, as illustrated in TABLE 3, the proposed MGC algorithm achieves better memory latency than all the other heuristic algorithms in most cases. On average, compared to MC, the memory latency reduces to 92.0% in the the first configuration, and 93.3% in the second configuration. Considering dynamic energy consumption, as illustrated in TABLE 4, MGC achieves lower dynamic energy consumption than all the other heuristic algorithms in most cases. On average, compared to MC, the memory latency reduces to 86.9% in the the first configuration, and 90.8% in the second configuration. Furthermore, it is shown that the results of MGC are very close to the *ILP allocator*. It is also found that in few cases MGC works

a little worse than the MC. The reason may lie in that, as a heuristic, MGC cannot guarantee better colorability than other heuristics in all cases.

6 Related work

Non-volatile memories (NVMs), with the features of low leakage power and high density, present new opportunities for addressing the memory leakage power consumption problem [19]. As stated above, compared with the traditional memory technologies such as SRAM and DRAM, write operations on NVMs suffer from considerably higher energy and longer latency. To overcome this shortcoming, lots of work has been done. [14] [11] [12] resort to hybrid memory architectures which exploits both NVMs and traditional memory technologies. [16] presents a way to reduce write activities on NVMs based main memory via small victim cache. [7] [9] propose compilation based methods to reduce write activities on NVMs.

7 Conclusion

In this paper, we propose a novel approach for static allocation of program variables into hybrid SPM based systems. Both an ILP formulation and a graph-coloring based algorithm are presented. The experimental results demonstrate that the proposed graph-coloring framework works well for the hybrid SPM allocation, and better than previous works.

References

[1] `http://www.microchip.com/en_US/family/8bit/index.html`.

[2] `http://www.lindo.com`.

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. In *ACM Transactions in Embedded Computing Systems*, volume 1, pages 6–26, 2002.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, 2002.

[5] X. Dong, N. P. Jouppi, and Y. Xie. Pcramsim: System-level performance, energy, and area modeling for phase-change ram. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 269–275, 2009.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *2001 IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[7] J. Hu, W.-C. Tseng, C. Xue, Q. Zhuge, Y. Zhao, and E.-M. Sha. Write activity minimization for nonvolatile main memory via scheduling and recomputation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):584 –592, april 2011.

[8] J. Hu, C. Xue, Q. Zhuge, W.-C. Tseng, and E.-M. Sha. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –6, march 2011.

[9] J. Hu, C. J. Xue, W.-C. Tseng, Y. He, M. Qiu, and E. H.-M. Sha. Reducing write activities on non-volatile memories in embedded cmps via data migration and recomputation. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 350–355, New York, NY, USA, 2010. ACM.

[10] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. the 2004 International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[11] J. Li, L. Shi, C. Xue, C. Yang, and Y. Xu. Exploiting set-level write non-uniformity for energy-efficient nvm-based hybrid cache. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pages 19 –28, oct. 2011.

[12] J. Li, C. Xue, and Y. Xu. Stt-ram based energy-efficiency hybrid cache for cmps. In *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, pages 31 –36, oct. 2011.

[13] L. Li, L. Gao, and J. Xue. Memroy coloring: A compiler approach for scratchpad memory management. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, 2005.

[14] T. Liu, Y. Zhao, C. Xue, and M. Li. Power-aware variable partitioning for dsps with hybrid pram and dram main memory. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 405 –410, june 2011.

[15] P. R. Panda, A. Nicolau, and N. Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publisher, Norwell, MA, USA, 1998.

[16] L. Shi, C. J. Xue, J. Hu, W.-C. Tseng, X. Zhou, and E. H.-M. Sha. Write activity reduction on flash main memory via smart victim cache. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, GLSVLSI '10, pages 91–94, New York, NY, USA, 2010. ACM.

[17] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. In *ACM Transactions in Embedded Computing Systems*, volume 5, pages 472–511, 2006.

[18] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, 1994.

[19] C. Xue, Y. Zhang, Y. Chen, G. Sun, J. Yang, and H. Li. Emerging non-volatile memories: Opportunities and challenges. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on*, pages 325 –334, oct. 2011.