Compiler-Assisted STT-RAM-Based Hybrid Cache for Energy Efficient Embedded Systems

Qingan Li, Jianhua Li, Liang Shi, Mengying Zhao, Chun Jason Xue, and Yanxiang He

***Abstract*— Hybrid caches consisting of static RAM (SRAM) and spin-torque transfer (STT)-RAM have been proposed recently for energy efficiency. To explore the advantages of hybrid cache, most of the management strategies for hybrid caches employ migration-based techniques to dynamically move write-intensive data from STT-RAM to SRAM. These techniques involve additional access operations, and thus lead to extra overheads. In this paper, we propose two compilation-based approaches to improve the energy efficiency and performance of STT-RAM-based hybrid cache by reducing the migration overheads. The first approach, migration-aware data layout, is proposed to reduce the migrations by rearranging the data layout. The second approach, migration-aware cache locking, is proposed to reduce the migrations by locking migration-intensive memory blocks into SRAM part of hybrid cache. Furthermore, experiments show that these two methods can be combined to reduce more migrations. The reduction of migration overheads can improve the energy efficiency and performance of STT-RAM-based hybrid cache. Experimental results show that, combining these two methods, on average, the number of write operations on STT-RAM is reduced by 17.6%, the number of migrations is reduced by 38.9%, the total dynamic energy is reduced by 15.6%, and the total access latency is reduced by 13.8%.**

***Index Terms*— Cache, compiler, hybrid cache, NVM, spin-torque transfer (STT)-RAM.**

I. Introduction

As TECHNOLOGY scaling continues, traditional static RAM (SRAM)-based caches are facing many challenges such as leakage power and scalability. Recent advancements in technology present spin-torque transfer (STT)-RAM as a new candidate for building caches [1]. Compared with SRAM, STT-RAM has higher storage density and negligible leakage power. Write operations on STT-RAM, however, have considerably longer latency and higher energy consumption than SRAM. To take advantage of both STT-RAM and SRAM, hybrid cache architectures (HCAs) have been studied in [2] and [3]. These studies show that caches built with multiple memory technologies can outperform its counterpart with single technology. To explore the advantages of the hybrid cache, migration-based techniques are commonly used to dynamically move write-intensive data from STT-RAM to SRAM [2]–[5]. Migrations require additional read and write operations for data movement and these additional access operations consume cycles as well as energy. This kind of overheads, called migration overheads, could degrade the energy efficiency and performance of the hybrid cache. For embedded systems, which often have more stringent power constraints, this is a big problem. To solve this problem, this paper proposes two migration-aware compilation-based approaches to improve the energy efficiency and performance of embedded systems with STT-RAM-based hybrid cache.

In this paper, a set of experiments are conducted and several observations are obtained. Motivated by these observations, this paper proposes two compilation methods, migration-aware data layout (MDL) and migration-aware cache locking (MCL) to reduce the number migrations by reducing the number of transition events. An R/W transition event is caused by a read operation followed by a write, or a write operation followed by a read, in a memory block. The MDL method is proposed to reduce the transition events by rearranging the data layout in the stack and static areas. The main concept is to place the data with consecutively same access operations into the same memory block to reduce transition events. The reduction of transition events leads to a reduction of migrations. The MCL method is proposed to reduce transition events by identifying transition-intensive memory blocks. By locking these memory blocks into the SRAM part of the hybrid cache and thus disabling the migration scheme for these memory blocks, a huge number of migrations can be reduced. The reduction of migrations can improve the energy efficiency and performance of the hybrid cache. The experimental results show that both MDL and MCL can improve the energy efficiency and performance of STT-RAM-based hybrid cache. Furthermore, it is found that greater improvement can be achieved by combining these two methods. On average, the number of write operations on STT-RAM is reduced by 17.6%, the number of migrations is reduced by 38.9%, the total dynamic energy is reduced by 15.6%, and the total access latency is reduced by 13.8%. The contributions of this paper include:

1) a set of experiments are conducted to analyze migrations in STT-RAM-based hybrid cache, and several key observations are presented;
2) an MDL method is proposed to improve the efficiency of STT-RAM-based hybrid cache;

Manuscript received November 1, 2012; revised May 22, 2013; accepted August 4, 2013. Date of publication August 30, 2013; date of current version July 22, 2014. This work was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region under Project CityU 123811 and Project 123210, and in part by the National Natural Science Foundation of China under Project 61170022 and Project 91118003.

Q. Li is with the School of Computer Science, Wuhan University, Wuhan, China, and also with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: ww345ww@gmail.com).

J. Li is with the School of Computer and Information, Hefei Institute of Technology, Hefei 230000, China (e-mail: jianhual@mail.ustc.edu.cn).

L. Shi is with the School of Computer Science, Chongqing University, Chongqing 400030, China (e-mail: shiliang@cqu.edu.cn).

M. Zhao and C. J. Xue are with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: my19900808@gmail.com; jasonxue@cityu.edu.hk).

Y. He is with the School of Computer Science, Wuhan University, Wuhan 430000, China (e-mail: yxhe@whu.edu.cn).

Digital Object Identifier 10.1109/TVLSI.2013.2278295

3) an MCL method is proposed to improve the efficiency of STT-RAM-based hybrid cache;
4) the MDL and cache locking methods are combined to improve the efficiency of hybrid cache further.

The rest of this paper is organized as follows. An analysis of migration overheads in STT-RAM-based hybrid cache is presented in Section II. The MDL method is presented in Section III, and the MCL method is presented in Section IV. Section V introduces a way of combining both the MDL and MCL methods to further improve the energy efficiency and performance of STT-RAM-based hybrid cache. In Section VI, a set of experiments are conducted to evaluate the propose methods for STT-RAM-based hybrid cache. The related works are discussed in Section VII. Finally, Section VIII concludes this paper.

II. Analysis of Migrations

In this section, the migration in STT-RAM-based hybrid cache is analyzed. This paper focuses on embedded systems, in which the configuration of one-level cache is often applied, such as Freescale e300 family [6] and Cortex-R-based MCUs [7]. For the evaluation, we consider a one-level on-chip data hybrid cache, where the cache size is 32-kbytes, the associativity is four-way (one way for SRAM and three ways for STT-RAM), and the cache-line size is 32 bytes. The cache management strategy in [5] is implemented for this evaluation. We defer the discussion of cache parameters to Section VI. From the evaluation, the following five key observations are obtained as follows.

1) Migrations are frequently triggered in STT-RAM-based hybrid cache, which shows significant migration overheads.
2) Migrations correlate closely with transition events in memory blocks.
3) Transition events can be reduced by rearranging the data layout.
4) In embedded systems, most of the transition events in memory blocks come from the stack and static areas, rather than the heap area.
5) Transition events from the stack and static areas are distributed unevenly over memory blocks.

Next, we will discuss these five observations in detail.

A. Overheads of Migrations

Migration-based techniques are proposed in recent works on hybrid caches [2]–[5]. For STT-RAM-based hybrid cache, the SRAM part is preferable for write operations and the STT-RAM part is preferable for read operations. Considering the high probability that a program writes data to a specific group of memory blocks repeatedly, data on STT-RAM should be migrated to SRAM if they are frequently written [2]. This kind of migration, triggered by write operations, is called w-migrate in this paper. Furthermore, it is observed that there is also high probability, which programs read data from a specific group of memory blocks repeatedly. Li *et al.* [5] proposed to migrate the memory blocks from SRAM into STT-RAM if they are read frequently. In this paper, this kind of migration, triggered by read operations, is called r-migrate. Both w-migrate and r-migrate require additional read and write operations for data movement, and thus consume extra cycles and energy. These additional overheads will degrade energy efficiency and performance of STT-RAM-based hybrid cache.

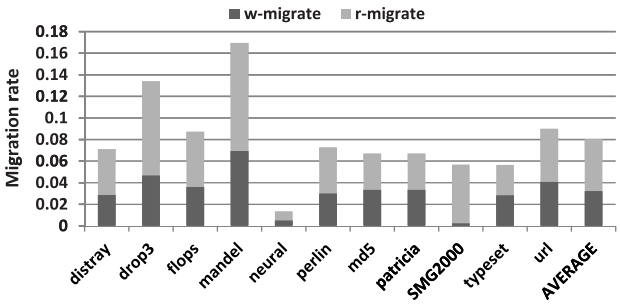


Fig. 1. Migration rate for the selected benchmarks. This rate is computed by dividing the number of migrations by the total number of memory accesses.

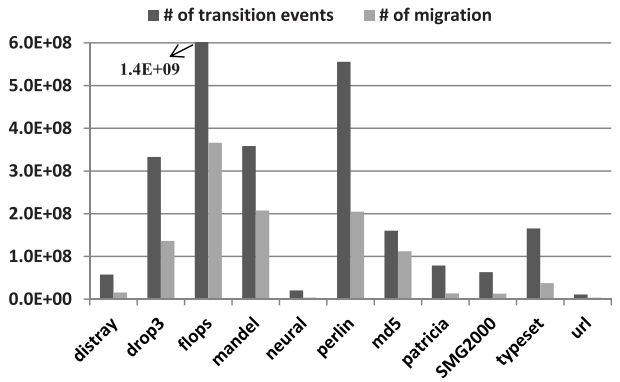


Fig. 2. Relationship between transition events and migrations.

Fig. 1 shows the number of migrations, consisting of w-migrate and r-migrate, for the selected benchmarks. It is found that, on average, eight migrations are triggered per hundred memory accesses. In other words, assuming each migration involves a read and a write operation, for each hundred memory accesses, about 16 additional operations are introduced by migrations. These overheads are detrimental to the system performance and energy consumption, and should be minimized.

B. Relationship Between Migrations and Transition Events

It is observed that the migration overheads correlate closely with transition events in access sequences. A R/W transition event is caused by a read operation followed by a write, or a write operation followed by a read, in a memory block. Fig. 2 shows the correlation between the number of migrations and the number of transition events in memory blocks. It is found that transition-intensive memory blocks are often migration-intensive memory blocks. The number of migrations changes along with the number of transition events in memory blocks. For the selected benchmarks, the correlation coefficient between the number of migrations and the number of transition events is about 0.94. Therefore, the number of transition events, which is visible at compilation

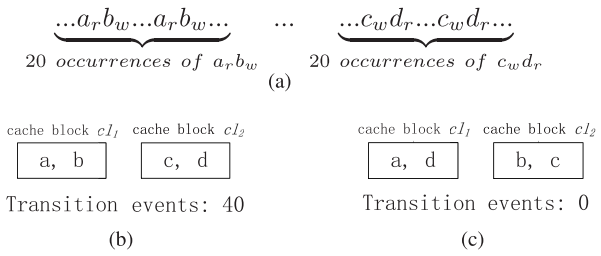


Fig. 3. Data layout for reducing transition events. (a) Access sequence with read and write information. (b) Default layout results in 40 transition events. (c) Better layout results in zero transition events.

time, can be used as a good indicator to reduce the migration overheads.

This significant correlation can be explained as follows. Generally, more transition events lead to more migrations. This is because, a transition event followed by several read operations (showing a trend of frequent read accesses) in an SRAM cache block triggers a r-migrate, and a transition event followed by several write operations (showing a trend of frequent write accesses) in an STT-RAM cache block triggers a w-migrate. Therefore, if too many transition events occur in memory blocks, the migration mechanism will be triggered frequently. In this situation, benefits from migration will be offset by the migration overheads.

C. Data Layout for Reducing Transition Events

Transition events from the stack and static areas can be reduced by rearranging the data layout, as shown in Fig. 3. Assume that there are four data objects, a, b, c, and d. For the access sequence in Fig. 3(a), with the default data layout, as shown in Fig. 3(b), both cache blocks will be written 20 times and read 20 times, hence the total number of transition events will be 40. However, if we assign a and d to cache block cl_1, b, and c to cache block cl_2, cl_1 will be read 40 times and cl_2 will be written 40 times. There will be no transition events in cache blocks, as shown in Fig. 3(c).

D. Distribution of Transition Events Over Memory Areas

Program data are stored in three memory areas: stack area, static area, and heap area. Local data, including local variables and compilation temporary variables, are stored in stack area. Stack area also includes the space for register protection, the space for parameters, and the space for return addresses. Stack storage can efficiently support a dynamic function invocation and provide space on demand. Global variables and static variables are stored in static area. Heap area includes dynamically allocated space (malloc function in C, or new operator in C++). Heap storage is commonly associated with the highest runtime cost among three kinds of storages, and thus it is rarely applied in low-end embedded systems [8], mostly due to the lack of a sophisticated operation system with memory management.

Fig. 4 shows the distribution of transition events over memory areas. It is found that for the selected benchmarks, on average, 81.0% of transition events come from the stack and static area. Considering the correlation between migrations and transition events, it shows that most of the migrations are from the stack and static areas. This paper focuses on reducing the migration overheads originating from the stack and static areas. Hence, the proposed approach may not work well for programs with lots of heap operations.

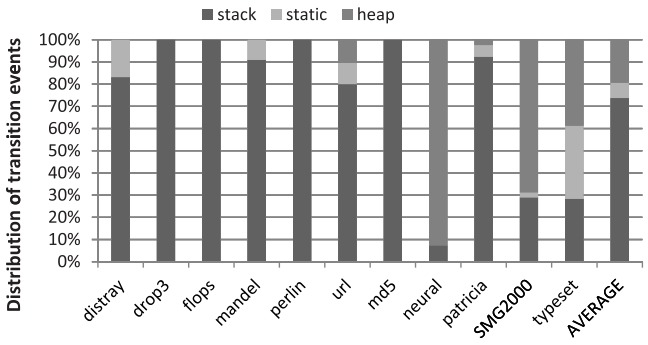


Fig. 4. Distribution of transition events over memory area.

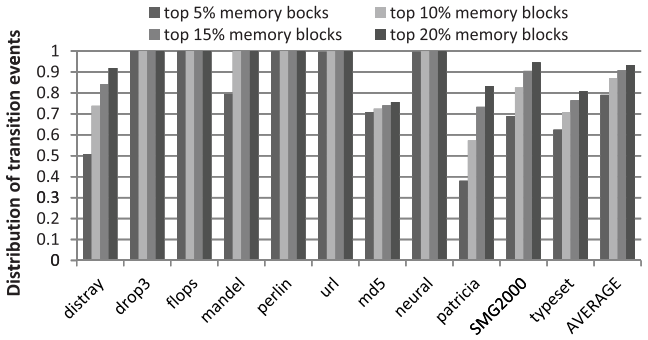


Fig. 5. Distribution of transition events over memory blocks in the stack.

E. Distribution of Transition Events Over Memory Blocks

Fig. 5 shows the distribution of transition events in the stack area over memory blocks. On average, 79.1% of transition events in the stack occur within the top 5% of memory blocks, and 93.3% of transition events in the stack occur within the top 20% of memory blocks. If we can statically identify these transition-intensive memory blocks at compilation time, and lock them in SRAM, then, no migrations are needed for these transition-intensive memory blocks. In other words, ideally we can eliminate 79.1% of migrations by locking 5% of memory blocks in SRAM. Furthermore, it is observed that the transition-intensive memory blocks are also write-intensive memory blocks. Therefore, locking memory blocks in SRAM can also make more write accesses occur in SRAM, which is preferable to write operations.

III. Migration-Aware Data Layout

This section introduces the MDL method to rearrange the data layout for the stack area and static area. As in most compilers, the data layout for the static area and the stack data layout for each function are conducted separately. The proposed approach rearranges the data layout for static area and stack frames in a similar fashion. Therefore, only data layout for stack frames is discussed in detail next. This method consists of three steps.

1) Obtain the data access sequence with data read/write information.
2) Assign the data into a set of memory blocks based on the data read/write operations. The size of a memory block equals the cache-line size.
3) Finalize the data layout according to the previous data assignment.

In the first step, we obtain the access sequence using static analysis, and collect the information about how many times two objects appear adjacent to each other in the access sequence. This information is encoded into a data proximity graph (DPG), and is used to guide the data assignment process. A DPG *G(V, E, w)* is a weighted graph, where *V* is the set of vertices and *E* is the set of edges. Each vertex represents a data object. For an edge *e(a, b)*, the weight *w(e)* is calculated by combining the occurrences of *a* and *b* appearing adjacent to each other as well as their corresponding read/write operation information in the access sequence. In the second step, we present a heuristic algorithm for the data assignment. In the weighted DPG, the weight of an edge shows the benefits from assigning the related pair of objects into the same memory block. The assignment splits the DPG into its vertex-induced subgraphs, where each subgraph corresponds to a memory block, such that the total weight of all subgraphs is maximized. In the last step, the data layout is finalized according to the previous data assignment. Note that the proposed algorithm does not handle objects of size greater than the cache-line size. These large objects are left to be placed using the default method.[1] In this paper, we assume that blocks are read or written to in nonoverlapping patterns.

A. Obtaining Access Sequence of Stack Objects

A static profiling technique [9] is employed in this paper to estimate the execution frequency of each statement and each basic block. Then, by visiting the control flow graph (CFG) annotated with execution frequency, the frequency that each pair of data objects appear adjacent to each other can be obtained. These pairs of consecutive data accesses can be categorized into four access patterns: read after read (RaR), write after write (WaW), write after read (RaW), and read after write (WaR). In the DPG, a larger weight value shows more benefits from assigning the related pair of data objects into the same memory block. Hence, the weight assignment process should treat access patterns in different ways.

1) For pattern RaR or WaW, assigning the related pair of data objects into the same memory block can improve the program locality, and thus reduce cache misses. To show this benefit, a positive weight α is assigned.
2) For pattern RaW or WaR, assigning the related pair of objects in the same memory block has double-edged effects, improving the program locality while at the same time increasing the number of transition events in this block. To show this benefit-cost tradeoff, a weight of $\alpha - \beta$ is associated with such a pattern. Here, α shows the benefit from program locality, whereas β shows the cost from increased migration overhead.

[1]By default, stack objects are placed in the order of declaration.

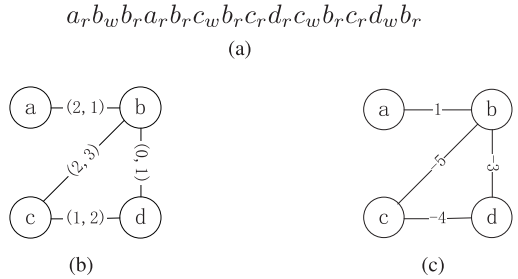


Fig. 6. Example for constructing the weighted DPG. (a) Memory access sequence. (b) Estimated data proximity. (c) Weighted DPG.

Now the benefit-cost tradeoff is determined by the choice of α and β, which will be explored in experiments evaluation. In Fig. 6, an example is presented to show the weight assignment process. Assume that there is a sequence of memory accesses, as shown in Fig. 6(a). The occurrences that two data objects appear adjacent to each other are shown in Fig. 6(b). In Fig. 6(b), the weight (x, y) of the edge (a, b) represents that the occurrences a and b appear adjacent to each other with the same operation is x, and the occurrences a and b appear adjacent to each other with different operations is y. The following equation is used for computing the weight value w: $w = \alpha x + (\alpha - \beta)y$. With $(2, 5)$ as coefficients of (α, β), the weighted DPG is built, as shown in Fig. 6(c).

B. Data Assignment

Data assignment is done by partitioning the DPG into its vertex-induced subgraphs such that the total weight value of all subgraphs are maximized, with the constraint that the total size of each subgraph is no more than the cache-line size. This partitioning problem can be easily formalized as an integer linear programming (ILP) problem. For scalability, a heuristic algorithm is also proposed.

1) ILP Formulation for Data Assignment: In this section, we present an ILP formulation for the problem of data assignment. There are three kinds of constraints for this problem. The assignment constraints ensure that each variable can be assigned into only one memory block. The exclusive constraints ensure that different variables cannot share the same address space. The alignment constraints ensure that the starting address of each variable should be aligned as required:

$B = \{b_1, \ldots, b_m\}$:	set of memory blocks;
P:	the size of each memory block in byte;
$V = \{v_1, \ldots, v_n\}$:	set of variables;
Q_i:	size of v_i in byte;
A_i:	alignment requirement of v_i;
$E_{i,j}$:	the weight of edge (v_i, v_j).

We use $x_{i,j}$ to describe whether variable v_i is assigned to memory block b_j, as defined in 1

$$x_{i,j} = \begin{cases} 1, & \text{if variable } v_i \text{ is assigned to block } b_j \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

Then, the assignment constraints can be formulated as (2)

$$\sum_{j=1}^{|B|} x_{i,j} = 1 \quad \forall i \in \{1, 2, \ldots, |V|\}. \quad (2)$$

Considering the alignment constraints for a variable v_i, the possible starting addresses of v_i is restricted to be multiples of the alignment number. We use w_i, as described in (3) for these constraints

$$\begin{cases} w_i \geq 0 \\ w_i \leq P/A_i - 1. \end{cases} \tag{3}$$

We use s_i to denote the starting address in the memory block assigned for variable v_i. If variable v_i is assigned to memory block b_j, then we can see that s_i must be in the range $[0,\ P - Q_i]$ and aligned as required. We can use the (4) to describe this constraint

$$\begin{cases} s_i \geq w_i \cdot A_i \\ s_i \leq P - Q_i. \end{cases} \tag{4}$$

For each pair of variables v_i and v_j, if they are assigned 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 show the order of such two variables, we define $y_{i,j}$ as (5)

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

Then, the exclusive constraints in the context of any pair of variables (v_i, v_j) can be formulated as (6). Notice that (6) must hold for any memory block b_k. If two variables are assigned into the same memory block b_t, i.e., $x_{i,t} + x_{j,t} = 2$, then (6) leads to $s_i + Q_i \leq s_j$ when $y_{i,j} = 1$, and $s_j + Q_j \leq s_i$ when $y_{i,j} = 0$. For any other memory block b_k, $x_{i,k} + x_{j,k} < 2$ holds, and thus 6 naturally holds. Therefore, (6) guarantees that different variables cannot share the same address space

$$\begin{cases} s_i + Q_i \leq s_j + (1 - y_{i,j}) \cdot P + (2 - x_{i,k} - x_{j,k}) \cdot P \\ s_j + Q_j \leq s_i + y_{i,j} \cdot P + (2 - x_{i,k} - x_{j,k}) \cdot P. \end{cases} \tag{6}$$

We use $z_{i,j}$ to show whether v_i and v_j are assigned into the same memory block, as described in (7). Therefore, $z_{i,j}$ must satisfy (8)

$$z_{i,j} = \begin{cases} 1, & \text{if } v_i \text{ and } v_j \text{ are in the same block} \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$$z_{i,j} \geq x_{i,k} + x_{j,k} - 1 \quad \forall k \in \{1, 2, \ldots, |B|\}. \tag{8}$$

The objective is to maximize the benefits, which can be formulated as follows:

$$\sum_{i=1}^{n} \sum_{j=1}^{m} z_{i,j} \cdot E_{i,j}. \tag{9}$$

2) Heuristic Algorithm for Data Assignment: A heuristic algorithm for data assignment is proposed, as shown in Algorithm 1. First, a list of empty memory blocks is built to be assigned. The size of each memory block equals the cache-line size. Then, the assignment process continues by iteratively retrieving the edge with the largest weight and assigning the related two data objects into the same memory block. After each assignment, the DPG is updated accordingly.

An example for the graph partitioning process is shown in Fig. 7. In the proposed algorithm, the weighted graph is a complete graph, but edges with weight less than or equal to zero are not shown for simplicity. It is assumed that all the data objects are of the same size, and each memory block (or memory block) can hold three objects. The graph to be partitioned is shown in Step 1 of Fig. 7. Because there are nine objects, the memory blocks list is initialized with three (9/3) empty blocks. In Step 2, the edge connected by nodes *a* and *b* has the largest weight, 100. Therefore, nodes *a* and *b* are assigned into the first memory block and they

Algorithm 1 Data Assignment by Graph Partitioning

Input:
graph[V][V]: the DPG
blocks: a empty list of memory blocks
nStackSize: stack size for the function
Output:
blocks: a list of assigned memory blocks

```
1: // Step 1: initialize the list of memory blocks;
2: int nThreshold = nStackSize/CACHE_LINE_SIZE;
3: for i = 1 to nThreshold do
4:    build a new memory block b and add it into blocks;
5: end for
6: // Step 2: assign data object into the list of memory blocks;
7: while graph is not empty do
8:    bool bSucc ← false;
9:    retrieve an edge e(v1, v2) from graph with the largest weight
      value (indicating benefits);
10:   // try to assign v1, v2 into the same block
11:   if both v1 and v2 are unassigned then
12:      if there is a block b can holding both v1 and v2 then
13:         bool bSucc ← true;
14:         assign v1 and v2 into the same block;
15:         merge vertex v1 and v2 in DPG;
16:      end if
17:   else if only v2 is unassigned then
18:      denote the block holding v1 as b;
19:      if b can hold v2 then
20:         bool bSucc ← true;
21:         assign this object into the block holding the other object;
22:         merge vertex v1 and v2 in DPG;
23:      end if
24:   end if
25:   // if this edge cannot guide a successful assignment, detete it;
26:   if ! bSucc then
27:      delete edge e(v1, v2) from DPG;
28:   end if
29: end while
30: // Step 3: assign the remaining data using the default method;
31: assign the remaining data objects using the default method
32: return blocks;
```

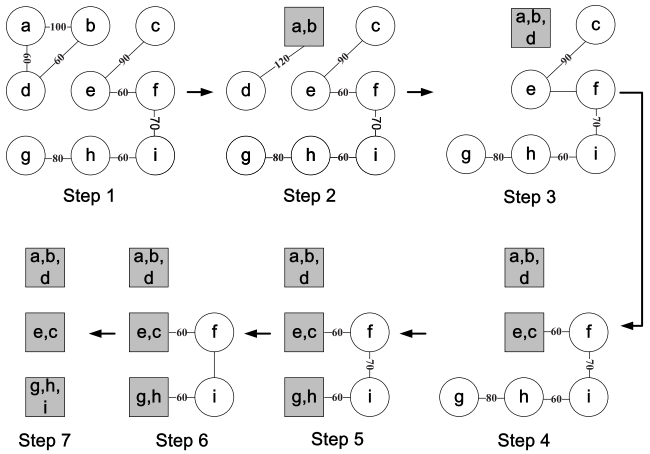


Fig. 7. Graph partitioning.

are merged into node (a, b). During the merging process, the edges connected by a and b are merged, too. For example, the two edges (a, d) and (b, d) are merged into (a, d), and the weight is updated $(120 = 60 + 60)$. In Step 3, the edge (a, d) has the largest weight, hence it is chosen and node d is merged with node (a, b). In Step 4, nodes e and c are assigned into the second memory block and they are merged. In Step 5, g and h are assigned into the third memory block. In Step 6, the edge (f, i) has the largest weight. None of the three memory blocks, however, can hold both f and i (Remember the assumption that each block can hold only three data objects). Therefore, this edge is deleted from the graph. In Step 7, f and i are assigned into the second and third memory blocks, respectively. Then, the data assignment process is completed.

C. Data Layout Finalization

After the graph partition process, a list of memory blocks is obtained. The offset of each data object internal to its memory block is also obtained. If the stack base pointer for each function is aligned with the cache-line size, we can conduct the finalization of stack layout by mapping the memory blocks into stack directly. In this paper, two tasks are carried out to align the stack base pointer with the cache-line size. First, extra instructions are inserted at the entry of the main function to align the stack base pointer of the main function with the cache-line size. Second, the stack size of each user function is expanded to be a multiple of the cache-line size. Assume that the cache-line size is C, and the original stack size of a function is x. After expansion, the new stack size for this function is: $\lceil x/C \rceil \times C$. After these two tasks, it is guaranteed that the stack base pointer of each user function invocation is aligned with the cache-line size. Note that there are some special stack contents, which cannot be rearranged freely. These contents include the space for saving registers and return address. Therefore, the mapping of memory blocks should be done after the space for all the special contents is already reserved.

IV. Migration-Aware Cache Locking

As observed in Section II-E, a small group of memory blocks often dominate the transition events in the stack. It is possible to identify the transition-intensive memory blocks at compilation time using static profiling techniques. If we can correctly identify these transition-intensive memory blocks and lock them in SRAM, migrations originating from these blocks could be eliminated. This section introduces the MCL method. This compilation-based method consists of three steps: 1) the information about which group of data objects will be loaded in the same memory block is analyzed at compilation time; 2) the transition-intensive memory blocks are identified based on static profiling; and 3) the transition-intensive memory blocks are locked in SRAM cache by employing cache locking instructions.

A. Identifying Data Objects Belonging to the Same Memory Block

At compilation time, we know each data object's offset relative to the stack base pointer, but cannot determine which group of data objects will be loaded into the same memory block during runtime. In other words, we need to know which group of data objects belongs to the same memory block. If the stack base pointer for each function is aligned with the cache-line size, it will be easy to correctly identify which group of data objects belongs to the same memory block according to the offset addresses relative to the stack base pointer. This alignment work can be done in the same way as detailed in Section III-C. After this alignment work, we can identify the data objects that belong to the same memory block. We only need to check whether the offsets of these two objects divided by the cache-line size are the same value.

B. Identifying the Transition-Intensive Memory Blocks

The static profiling technique proposed in [9] can be used to estimate the number of transition events between the data objects. This technique can estimate the execution frequency of each statement, each basic block and each control flow edge by exploring heuristics. With this technique, the following information can be obtained at compilation time: the execution frequency of each basic block b, denoted as **Bfreq**(b), and the frequency from a block b to its succeeding block c, denoted as **Efreq**(b, c). The estimation of transition events for memory blocks consists of two parts, as shown in Algorithm 2. First, the CFG is visited, and within each basic block, the adjacent accesses to the same memory block are checked (from lines 3 to 24).

When a pair of consecutive data accesses to the same memory block is found to be in different operation modes, the contribution to transition events from this pair of accesses, **Bfreq**(b), is acknowledged (from lines 17 to 20). Second, the CFG is revisited, and across basic blocks, the adjacent accesses to the same memory block are checked (from lines 26 to 34). When a pair of consecutive data accesses to the same memory block is found to be in different operation modes, the contribution to transition events from this pair of accesses, **Efreq**(b, b_1), is acknowledged (from lines 29 to 31).

By now, the number of transition events in each memory blocks is computed. We need to determine, which memory blocks are transition-intensive and should be locked in SRAM. In the proposed approach, a threshold value N is used for this decision. If the number of transition events in a memory block mb is greater than N, then, mb will be identified as a transition-intensive memory block and will be locked in SRAM. The choice of N will affect the performance of the proposed MCL method. If N is too small, too many memory blocks will be locked, and only a small group of cache lines is available for cache replacement. Thus, the hit ratio of the hybrid cache will be hindered. If N is too large, few memory blocks will be locked, then the proposed technique will not be effective. The choice of N will be discussed in detail in Section VI.

Algorithm 2 Estimating the Number of Transition Events in Each Memory Block

Input:
CFG: the CFG of a function
hFirstAcc: memory block → basic block → data access, storing the first data access in each basic block to a memory block
hLastAcc: memory block → basic block → data access, storing the last data access in each basic block to a memory block
Output:
transMap: a map storing the number of estimated transition events for each memory block

```
initialize the weight of each element in transMap to be zero;
// Part 1: check pairs of accesses within a basic block
for each basic block b in the CFG do
    // DA(v, o) represents a data access: v represents a data object, o represents the access type, read or write
    // da_0 is used to record the previous data access
    // hPreAcc: memory block → data access, storing the previous data access in this basic block to a memory block
    DA da_0(v_0, o_0);
    for each statements s in basic block b do
        for each data access da_1(v_1, o_1) in statement s do
            int blockID ← v_1.offset;
            // if da_1 is the first data access in b to blockID
            if hPreAcc[blockID] is empty then
                hPreAcc[blockID] ← o_1;
                hFirstAcc[blockID][b] ← o_1;
                continue;
                // if da_1 accesses blockID differently from the previous access to blockID
            else if hPreAcc[blockID] ≠ o_1 then
                hPreAcc[blockID] ← o_1;
                transMap[blockID] += Bfreq(b);
            end if
            hLastAcc[blockID][b] ← o_1
        end for
    end for
end for
// Part 2: check pairs of accesses acrosss basic blocks
for each memory block blockID do
    for each basic block b in the CFG do
        for each succeeding basic block b_1 of b do
            if hLastAcc[blockID][b] ≠ hFirstAcc[blockID][b_1] then
                transMap[blockID] += Efreq(b, b_1);
            end if
        end for
    end for
end for
return true;
```

C. Locking Transition-Intensive Memory Blocks in SRAM

There are many different ways that we can implement the cache locking of transition-intensive memory blocks. A potential implementation method is shown in this section. Many microcontrollers (MCUs) support functions of data prefetching and cache locking [6], [7]. In these MCUs, the MCL method can be implemented without any modification of the hardware. When the control flow enters a function, the transition-intensive memory blocks can be prefetched into the SRAM part of the STT-RAM-based hybrid cache (if the target SRAM location is already locked, unlock it first), and then these cache lines can be locked using the cache locking function to avoid being evicted by cache replacement strategies and being migrated to STT-RAM. Note that in this paper, the MCL method is a preemptive approach. It means, when the execution enters a function, its transition-intensive memory blocks can preempt the SRAM cache lines locked by previous functions (via the unlocking function). The detail is shown in Algorithm 3. These instructions for prefetching, cache locking and cache unlocking should be inserted at the entry of each function, and thus will be executed before any other instructions of this function.

Algorithm 3 CacheLock: Cache Locking Transition-Intensive Memory Blocks into SRAM

Input:
blocks: the list of transition-intensive memory block of a function

```
for each memory block b in blocks do
    // denote block b's target location in SRAM as cache line c
    // Step 1: cache unlocking
    if c is already locked then
        insert instructions to unlock cache line c;
    end if
    // Step 2: memory block pre-fetching
    insert instructions to pre-fetch memory block b into cache line c;
    // Step 3: cache locking
    insert instructions to lock cache line c;
end for
return true;
```

V. Combining MDL and MCL

The idea of combining MDL and MCL together is motivated by two aspects. First, by the MDL method, the total number of transition events in memory blocks can be reduced. Second, as stated in Section II, migrations are sensitive to transition events in memory blocks, which can be changed by rearranging data layout. If we can rearrange the data layout in a way that the distribution of transition events is more concentrated over memory blocks, the effectiveness of the MCL method will be enhanced. This is because that, with a more concentrated distribution of transition events, more migrations can be eliminated by cache locking less memory blocks. This section proposes a method to reduce migration overheads by combining MDL and MCL. This method, called migration-aware data layout and cache locking (MDCL), consists of five steps.

1) Obtain the data access sequence using static analysis and collect information about read/write operations, including transition events. This step is similar to Section III-A.
2) Assign data objects into memory blocks based on the information about read/write operations. Exactly like the MDL method, this step aims to reduce the total number of transition events in memory blocks. Hence, this step is similar to Section III-B. However, as will be discussed later, the best (α, β) coefficients for MDCL may differ from that for MDL (Section III-B).
3) Finalize the data layout. This step is similar to Section III-C.

4) Identify the transition-intensive memory blocks according to transition events and the new data layout. This step is similar to Section IV-B.
5) Lock transition-intensive memory blocks into SRAM of STT-RAM-based hybrid cache. This step is similar to Section IV-C.

Note that for the second step, the best data layout for MDCL may differ from that for MDL (Section III-B), and thus the best (α, β) coefficients may differ. This is because, the assignment in MDCL pursues three goals: improving program locality, reducing the total number of transition events, and clustering the remaining transition events into a small number of memory blocks (because the following cache locking method will be conducted based on this data layout), while the assignment in MDL in Section III-B considers only the first two goals.

VI. Experiment

In this section, the experimental setup is introduced first. Then, the experimental results for evaluating the proposed methods are presented. After that, the parameters involved in the proposed methods, including the coefficients (α, β) for weight assignment and the threshold value N for selecting transition-intensive memory blocks, are explored. Finally, the overheads of these methods are discussed.

TABLE I
Benchmarks Characteristics

program	read	write	description
distray	1.7E+09	7.9E+08	ray tracing
drop3	4.5E+08	3.8E+08	bit stream manipulating
flops	3.3E+09	8.2E+08	estimating MFLOPS rating
mandel	6.8E+08	5.0E+08	drawing Mandelbrot/Julia set
md5	1.4E+09	2.6E+08	MD5 algorithm
neural	2.7E+08	1.1E+07	storing in Hopfield network
patricia	1.4E+08	9.4E+07	PATRICIA trie implementation
perlin	2.5E+08	3.2E+07	noise algorithm
smg2000	6.2E+09	1.2E+09	a multigrid solver
typeset	2.4E+08	1.6E+08	document formatting system
url	4.5E+09	3.3E+07	URL switching

TABLE II
Architecture Parameters

Parameter	Value
processor	single core
hybrid data cache	32KB, 32B line size, LRU, 4-way 1-way for sram (8KB) 3-way for stt-ram (24KB) write allocation, write back sram access latency: 6 cycles sram access dynamic energy: 0.388 nJ stt-ram read/write latency: 6/28 cycles stt-ram read/write dynamic energy: 0.4/2.3 nJ
main memory	latency: 300 cycles

A. Experimental Setup

The proposed compilation technique is implemented on LLVM [10]. The benchmarks as well as their input files are selected from the LLVM test suites. The characteristics of these benchmarks are shown in Table I. Four groups of executables are generated and the related statistics are collected. The first group, called the reference group (RG), is compiled by LLVM compiler using the default method for data layout. This group is used as the baseline for comparison. The second group is compiled with the MDL method. The third group is compiled with the MCL method. The fourth group is compiled with the MDCL method. All groups are compiled with O3 optimization.

A pin-based [11] cache simulator is developed for evaluating the proposed methods. This simulator is implemented with the cache management strategy in [5]. Note that the strategy in [5] targets chip multiprocessors, but this paper focuses on embedded systems with single-core processors, and thus the intercore migration strategy in [5] is excluded. The target architecture is shown in Table II. The cache parameters and memory parameters are obtained from a modified CACTI [12].

B. Comparison of the Proposed Methods

This section presents the comparison of the proposed techniques with the work proposed in [5]. There are several transactions that directly affect the total cost of memory accesses. Among these transactions, the most important ones are shown in Table III. In this table, the second column shows the atomic transactions related to each transaction in the first column, where MR/MW represents STT-RAM read/write, SR/SW represents SRAM read/write, and MMR/MMW represents main memory read/write. Our experiments evaluate the influences of the proposed technique on these transactions. Here, we choose 25 as the threshold N for identifying the transition-intensive cache lines and $(0, 4)$ as the (α, β) configuration for rearranging data layout. The experimental results are normalized to the baseline of RG.

Fig. 8(a) shows that compared with RG, the number of migrations is reduced by 26.6% in MCL, 12.0% in MDL, and 38.9% in MDCL on average. Fig. 8(b) shows that compared with RG, the number of write operations on STT-RAM is reduced by 8.4% in MCL, 10.1% in MDL, and 17.6% in MDCL on average. These decrements can significantly save cost because write operations in STT-RAM have longer latency and higher energy consumption than SRAM.

As discussed above, the proposed technique could reduce the number of migrations, and the number of write accesses to STT-RAM. Therefore, it is promising that the proposed technique could improve the efficiency of the STT-RAM-based hybrid cache. The total latency is evaluated using the cache and memory parameters shown in Table II. Equation (10) is used to estimate the total latency, where $MR_{\text{num}}/MW_{\text{num}}$ represents the number of STT-RAM read/write transactions, $MR_{\text{cost}}/MW_{\text{cost}}$ represents the cost (latency) per STT-RAM read/write transaction. The total dynamic energy consumption is estimated in a similar fashion. We only consider dynamic energy consumed by the cache, but not dynamic energy consumed by the main memory. As shown in Fig. 8(d), compared with RG, the total dynamic energy is reduced by 8.6% in MCL, 6.9% in MDL, and 15.6% in MDCL. As shown in Fig. 8(c), the total latency is reduced by

TABLE III
IMPORTANT TRANSACTIONS

Transaction	Atomic transactions	Detail
Read Hit	MR/SR	read on STT-RAM/SRAM
Write Hit	MW/SW	write on STT-RAM/SRAM
Read Miss	MMR + MW + MR	fetch the targeted memory block from main memory into STT-RAM and read it
Write Miss	MMR + SW + SW	fetch the targeted memory block from main memory into SRAM and write it
R-migrate	SR + MW	migrate a memory block from SRAM to STT-RAM
W-migrate	SR + SW + MR + MW	exchange two memory blocks in SRAM and STT-RAM
Write Back	MR + MMW	write a memory block in STT-RAM back to main memory[2]

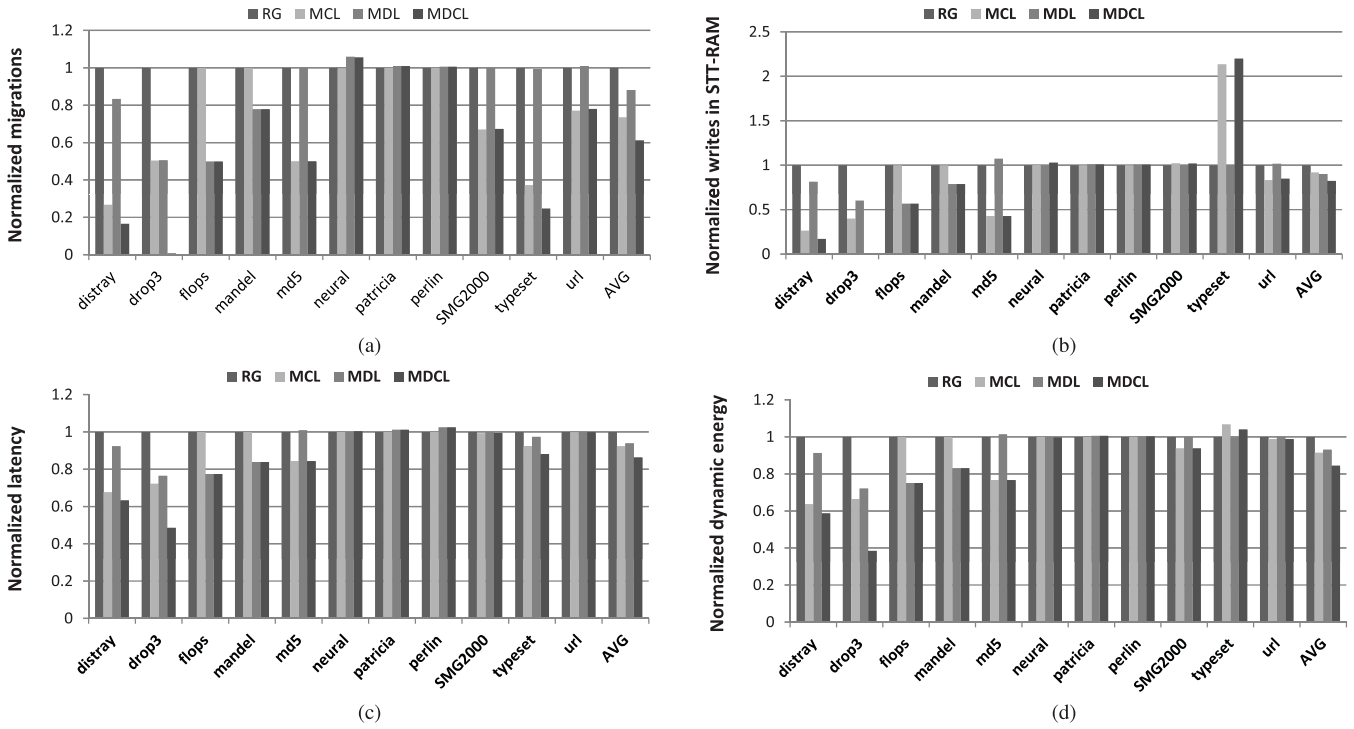


Fig. 8. Comparison of results. The (α, β) configuration is (0, 4). The N is 25. All results are normalized to results of the RG. (a) Reduction of migrations. (b) Reduction of writes in STT-RAM. (c) Reduction of total latency. (d) Reduction of total dynamic energy.

7.7% in MCL, 6.2% in MDL, and 13.8% in MDCL on average.

$$\begin{aligned}\text{total\_latency} = {} & MR_{\text{num}} \cdot MR_{\text{cost}} + MW_{\text{num}} \cdot MW_{\text{cost}} + SR_{\text{num}} \\ & \cdot SR_{\text{cost}} + SW_{\text{num}} \cdot SW_{\text{cost}} + MMR_{\text{num}} \\ & \cdot MMR_{\text{cost}} + MMW_{\text{num}} \cdot MMW_{\text{cost}}. \end{aligned} \tag{10}$$

C. Efficiency of the Proposed Methods

There are three observations from the experimental results: 1) for most of the benchmarks, it can improve the efficiency; 2) for several benchmarks, it works very well; and 3) it does not work well for some benchmarks. In rare cases, such as neural, it has a negative effect, although the negative effect is very limited. Here, we present discussions about the efficiency of the proposed methods.

1) Efficiency of MDL: For the proposed data layout method, the observations stated above are mainly due to five reasons. First, less migrations can be reduced if less transition events occurs in the stack and static areas, because the proposed approach focuses on these two areas. This explains why the MDL method does not work well for neural, SMG2000, and typeset (see Fig. 4).

Second, the data layout method works in a granularity of memory block size. If the stack frame size is relatively small, there is a little opportunity to reduce the transition events by rearranging the data layout. This explains why the MDL method does not work well for patricia, perlin, and url.

Third, the more concentrated the original distribution of transition events (over memory blocks) is, the more benefits MDL method achieves, because MDL method focuses on reducing the transition events. This explains why MDL method works well for drop3, flops, and mandel (see Fig. 5).

Fourth, the static profiling technique works well for most of the benchmarks, but it cannot guarantee a precision estimation of transition events, and thus cannot guarantee an optimal data layout. This explains why the MDL method does not work well for neural. The original data layout of neural seems to be optimal with regard to migration reduction.

[2]The management strategy for hybrid cache in [5] forces write back comes from STT-RAM.

TABLE IV

OVERHEADS FROM CACHE LOCKING IN MCL AS N VARIES. FOR EACH BENCHMARK, THE VALUE BEFORE/IS THE SUM OF F_{Lines}, SHOWING THE CODE SIZE OVERHEAD. THE VALUE AFTER/IS THE SUM OF $F_{\text{Lines}} \cdot F_{\text{Exec}}$, SHOWING THE RUNTIME OVERHEAD

N	*distray*	*drop3*	*flops*	*mandel*	*md5*	*neural*	*patricia*	*perlin*	*SMG2000*	*typeset*	*url*
10	10/68949301	2/40	2/2	3/3	4/31998280	4/62	0/0	2/2	227/749446	116/2371074	4/160900
25	9/68949300	2/40	2/2	3/3	4/31998280	1/30	0/0	2/2	225/749442	116/2371074	3/160500
10^2	9/68949300	1/20	0/0	2/2	4/31998280	0/0	0/0	2/2	220/749433	116/2371074	3/160500
10^3	9/68949300	0/0	0/0	1/1	2/15999140	0/0	0/0	0/0	215/749420	113/2371033	2/159600
10^4	8/68949299	0/0	0/0	0/0	0/0	0/0	0/0	0/0	212/749410	112/2363050	0/0
10^5	8/68949299	0/0	0/0	0/0	0/0	0/0	0/0	0/0	209/749314	109/2357365	0/0
10^6	8/68949299	0/0	0/0	0/0	0/0	0/0	0/0	0/0	194/748650	103/2346619	0/0
10^7	8/68949299	0/0	0/0	0/0	0/0	0/0	0/0	0/0	167/747622	98/2332588	0/0
10^8	5/45397547	0/0	0/0	0/0	0/0	0/0	0/0	0/0	159/747400	97/2332585	0/0

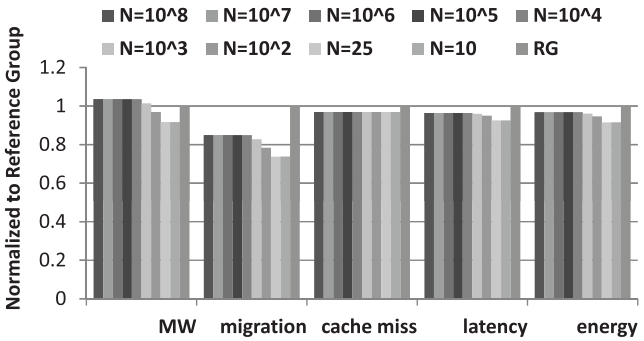


Fig. 9. Choice of N. MW: write accesses in STT-RAM. The data are average values over all selected benchmarks.

Fifth, only considering pair-wise proximity relationship between the data objects limits the effectiveness of data layout.

2) Efficiency of MCL: The reasons affecting the efficiency of the MDL method can affect the MCL method in a similar fashion. Let us review the reduction of migrations, as shown in Fig. 8(a).

For distray, drop3, flops, mandel, md5, and SMG2000, the results are reasonable. The MCL method can reduce the number of migrations. The MDCL method brings greater improvement, because the number of migrations is reduced first by the MDL method, and then by the MCL method.

For neural, patricia, perlin, and url, the reduction of migrations by MCL is not good. The reasons are the same as those stated above in the MDL method. In addition, no transition-intensive memory block is identified for the patracia benchmark, as shown in Table IV.

For typeset, the MCL method can reduce a large number of migrations. However, it increases the number of writes in STT-RAM greatly, which is mainly due to the large memory footprint of typeset. For this benchmark, a large number of SRAM cache lines have been locked and cannot be vacated for other memory accesses. Thus, lots of memory writes trigger STT-RAM writes.

D. Choice of N

In this section, we discuss the choice of the threshold N, which is used to identify the transition-intensive memory blocks. As shown in Fig. 9, as N becomes smaller, the number of migrations and writes in STT-RAM decrease. This is because, as N becomes smaller, more memory blocks will be identified as transition-intensive. The increment of locked memory blocks may enhance the benefits from the MCL method. Thus, the improvement may be better with a small N. However, it does not hold all the time for two reasons. First, as N becomes smaller, more SRAM cache lines are locked, and thus STT-RAM cache lines may be very hot with both read and write accesses. The frequent use of STT-RAM cache lines with write operations is decremental to the system performance and energy efficiency. The experimental results for typeset is an example. Second, as N becomes smaller, more memory blocks are locked, and thus the overheads from cache locking become larger, as discussed in Section VI-F.

E. Choice of (α, β) Coefficients

The choice of (α, β) coefficients can affect the weight assignment process for rearranging data layout. Hence, it has significant influence on the effectiveness of the MDL and MDCL methods. Although our experiments show that $(0, 4)$ is a good choice, there is a need to explore more configurations.

1) The Choice of (α, β) in MDL: Here, we discuss the choice of the (α, β) coefficients in the MDL method. Nine configurations are studied and the results are shown in Fig. 10. It is found that seven of the nine configurations show improvements over the base line. It is also observed that, to obtain the best results, the value of α should be small relative to the value of β. Because a larger α relative to β shows more weight from program locality, the experimental results in Fig. 10 show that, it is not enough to consider only program locality during the data layout process. The cost from migrations should also be considered. In addition, (0, 4) works best among all the configurations. This is mainly because, the variation of (α, β) has little impact on the program locality, while the migration overhead can be significantly reduced with a big β relative to α, because the weight for RaW and WaR, α–β, is too small.

2) The Choice of (α, β) in MDCL: Fig. 11 shows the influence of this choice in MDCL when N is 25. It is found that, the same as in the MDL method, $(0, 4)$ performs the best again. Furthermore, the number of migrations can be greatly reduced and the performance and energy efficiency can be improved with other choices of (α, β) as well.

F. Overheads of the Proposed Methods

Both MDL and MCL methods need to align the stack base address for each function with the cache-line size. This alignment work may lead to an increase of stack size.

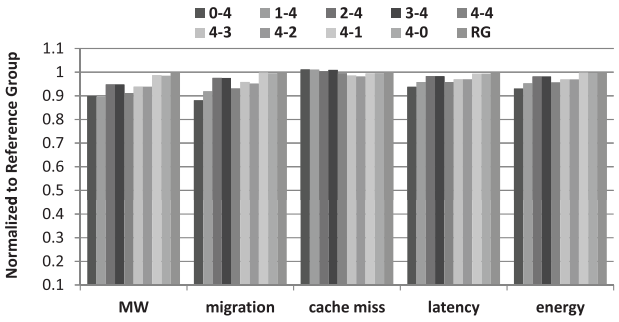


Fig. 10. Choice of (α, β) in MDL. MW: write accesses in STT-RAM. The data are average values over all selected benchmarks.

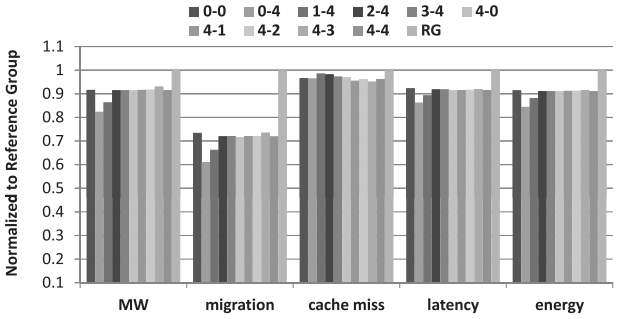


Fig. 11. Choice of (α, β) in MDCL with N is 25 MW: write accesses in STT-RAM. (0, 0) shows the MCL method based on the original data layout. The data are average values over all selected benchmarks.

These overheads originate from two aspects. First, there are space overheads during aligning the initial stack address with the cache-line size. These overheads are limited by the cache-line size. Second, there are space overheads during adjusting the stack size of each function to be a multiple of the cache-line size. These overheads are limited by CACHE_LINE_SIZE · length, where length is the number of functions in the longest function calling chain during runtime. In addition, this paper may also lead to the increase of code size, since extra instructions are needed to align the initial stack address with the cache-line size for the main function. However, not more than four instructions are needed for this task. Therefore, the space overheads as well as the runtime overheads from these instructions are negligible.

The MCL method results in additional overheads. These overheads mainly come from the implementation of locking transition-intensive memory blocks. Assume that the number of transition-intensive memory blocks of a function f is f_{lines}, and f executes f_{exec} times. Then, for function f, the code size overheads are linear to f_{lines}; the runtime overheads are linear to $f_{\text{lines}} \cdot f_{\text{exec}}$. Table IV shows these overheads. The value before / is the sum of f_{lines} for each benchmark, and the value after / is the sum of f_{exec} for each benchmark. A smaller N leads to more overheads. It is found that, the worst case benchmark SMG2000 needs to insert at most of the 227 prefetch and cache locking instructions. Therefore, the code size overheads are negligible. In addition, the worst case benchmark distray needs to execute at most of the 69 million of prefetch and cache locking instructions for cache locking. This amounts to only 0.3% of performance overhead, considering the number of instruction counts, and memory accesses are significantly larger (see Table I). Therefore, the runtime overheads are also negligible.

VII. Related Work

Recently researchers have been attracted by several new memory technologies, including phase-change RAM [13]–[15], and STT-RAM [16], [17]. There are several studies architecting energy efficient STT-RAM-based hybrid cache. Sun *et al.* [2] proposed hybrid caches consisting of SRAM and S-TT-RAM, and employed migration-based policy to mitigate the drawbacks of STT-RAM. Wu *et al.* [18], [3] evaluated two types of HCAs, including inter cache level HCA and intracache level HCA, both of which employ migration policy. Their experiments show that these HCAs can improve the energy efficiency and performance. More migration-based policies were explored in [4] and [5] to further improve the efficiency of STT-RAM-based hybrid cache. These work proposed architectural level technologies to improve the efficiency of hybrid caches, while this paper proposes compilation techniques for the same goal.

The compilation-based techniques for cache aware data placement have been studied for a long time. A general framework for cache conscious data placement was proposed in [19]. Petrank *et al.* [20] presented a good survey on cache aware data placement. These works focused on improving the program locality for pure SRAM-based caches to improve the performance, while our work considers read/write asymmetry together with locality to improve efficiency of STT-RAM-based hybrid caches.

Recently, Li *et al.* [21] proposed a software approach to improve the efficiency of STT-RAM-based hybrid cache. Through identifying the write reuse within data structures, their method proposed to predispatch instructions in code prior to write accesses to notify the CPU to perform migration operations. This method needs to implement extra instructions to explicitly specify the cache migration operations. Therefore, this method involves a complicated hardware design, while our work requires no modification of hardware.

VIII. Conclusion

Most of the works on STT-RAM-based hybrid cache employ migration-based policies to mitigate the high cost of write operations on STT-RAM and to improve the efficiency. In this paper, a set of experiments are conducted to analyze the migrations, and several observations are obtained. With these observations, two compilation approaches, MDL and MCL, are proposed. Experiments show that the proposed approaches can improve the energy efficiency and performance of STT-RAM-based hybrid cache. Furthermore, these two approaches can work together to improve the efficiency further.

References

[1] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement," in *Proc. 45th Annu. Design Autom. Conf.*, 2008, pp. 554–559.

[2] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *Proc. IEEE 15th Int. Symp. HPCA*, Feb. 2009, pp. 239–249.

[3] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 34–45.

[4] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad, "High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement," in *Proc. 17th IEEE/ACM Int. Symp. Low-Power Electron. Design*, Aug. 2011, pp. 79–84.

[5] J. Li, C. Xue, and Y. Xu, "STT-RAM based energy-efficiency hybrid cache for CMPs," in *Proc. IEEE/IFIP 19th Int. Conf. VLSI-SoC*, Oct. 2011, pp. 31–36.

[6] (2007). *e300 Power Architecture Core Family Reference Manual* [Online]. Available: http://www.freescale.com/files/32bit/doc/ref_manual/e300coreRM.pdf

[7] (2013). *Cortex-R Series* [Online]. Available: http://www.arm.com/products/processors/cortex-r/index.php

[8] (2010). *Do you use or Allow Dynamic Memory Allocation in Your Embedded Design?* [Online]. Available: http://www.embeddedinsights.com/

[9] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proc. 27th Annu. Int. Symp. Microarchit.*, 1994, pp. 1–11.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generat. Optim., Feedback-Directed Runtime Optim.*, 2004, pp. 1–12.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2005, pp. 190–200.

[12] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2007, pp. 3–14.

[13] W.-C. Tseng, C. J. Xue, Q. Zhuge, J. Hu, and E.-M. Sha, "Optimal scheduling to minimize non-volatile memory access time with hardware cache," in *Proc. 18th IEEE/IFIP VLSI-SoC*, Sep. 2010, pp. 131–136.

[14] T. Liu, Y. Zhao, C. Xue, and M. Li, "Power-aware variable partitioning for DSPs with hybrid PRAM and DRAM main memory," in *Proc. DAC*, 2011, pp. 405–410.

[15] 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," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 584–592, Apr. 2011.

[16] 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 *Proc. 9th IEEE Symp. ESTIMedia*, Oct. 2011, pp. 19–28.

[17] J. Li, L. Shi, Q. Li, C. J. Xue, Y. Chen, and Y. Xu, "Cache coherence enabled adaptive refresh for volatile STT-RAM," in *Proc. DATE*, Mar. 2013, pp. 1247–1250.

[18] X. Wu, J. Li, L. Zhang, E. Speight, and Y. Xie, "Power and performance of read-write aware hybrid caches with non-volatile memories," in *Proc. Conf. Design, Autom. Test Eur.*, 2009, pp. 737–742.

[19] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proc. 8th Int. Conf. Archit. Support Program. Lang. Operat. Syst.*, 1998, pp. 139–149.

[20] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement," *Nordic J. Comput.*, vol. 12, pp. 275–307, Jun. 2005.

[21] Y. Li, Y. Chen, and A. K. Jones, "A software approach for combating asymmetries of non-volatile memories," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, Aug. 2012, pp. 191–196.

Qingan Li received the B.S. degree in computer science from Wuhan University, Wuhan, China, in 2008, where he is currently pursuing the Ph.D. degree in computer software and theory.

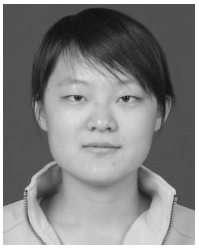
His current research interests include compiler optimization, program analysis, and embedded systems.

Jianhua Li received the B.S. degree in computer science from Anqing Teachers' College, Anhui, China, in 2007, and the Ph.D. degree in computer software and theory from the University of Science and Technology of China, Anhui, in 2013.

He is currently a Lecturer with the School of Computer Science and Information, Hefei Institute of Technology, Anhui. His current research interests include on-chip networks, multicore memory system, and emerging nonvolatile memories.

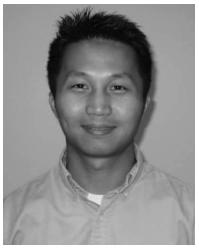
Liang Shi received the B.S. degree in computer science from the Xi'an University of Posts and Telecommunications, Xi'an, China, in 2008, and the Ph.D. degree in computer science from the University of Science and Technology of China, Anhui, China, in 2013.

He is currently a Full-Time Teacher with the School of Computer Science, Chongqing University, Chongqing, China. His current research interests include flash memory, embedded systems, and emerging nonvolatile memory technology.



Mengying Zhao received the B.S. degree in computer science from Shandong University, Shandong, China, in 2011. She is currently pursuing the Ph.D. degree in computer science from City University of Hong Kong, Hong Kong.

Her current research interests include embedded and real-time systems, architecture, and memory optimizations.



Chun Jason Xue received the B.S. degree in computer science and engineering from the University of Texas at Arlington, Arlington¸ TX, USA, in 1997, and the M.S. and Ph.D. degrees in computer science from the University of Texas at Dallas, Dallas, TX, USA, in 2002 and 2007, respectively.

He is currently an Assistant Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His current research interests include embedded systems, nonvolatile memory, real time systems, and hardware/software co-design.

Yanxiang He received the B.S. and M.S. degrees in mathematics, and the Ph.D. degree in computer science from Wuhan University, Wuhan, China, in 1973, 1975, and 1999, respectively.

His current research interests include trustworthy software engineering, mobile computing, and distribution computing.