Compiler-Assisted Refresh Minimization for Volatile STT-RAM Cache

Qingan Li, Yanxiang He, Jianhua Li, Liang Shi, Yiran Chen, *Member, IEEE*, and Chun Jason Xue

Abstract—Spin-transfer torque RAM (STT-RAM) has been proposed to build on-chip caches because of its attractive features such as high storage density and ultra low leakage power. However, long write latency and high write energy are the two challenges for STT-RAM. Recently, researchers propose to improve the write performance of STT-RAM by relaxing its non-volatility property. To avoid data losses resulting from volatility, refresh schemes have been proposed. However, refresh operations consume additional overhead. In this paper, we propose to significantly reduce the number of refresh operations through re-arranging program data layout at compilation time. An *N-refresh* scheme is also proposed to further reduce the number of refreshes. Experimental results show that, on average, the proposed methods can reduce the number of refresh operations by 84.2 percent, and reduce the dynamic energy consumption by 38.0 percent for volatile STT-RAM caches while incurring only 4.1 percent performance degradation.

Index Terms—Compilation, volatile STT-RAM, refresh

1 Introduction

As technology scales down, traditional SRAM-based caches face challenges such as leakage energy and scalability. Recent advancements in memory technology present spin-transfer torque RAM (STT-RAM) as a new candidate for building caches [1]. Compared to SRAM, STT-RAM has higher storage density and negligible leakage power. However, compared to SRAM, write operations in STT-RAM have longer latency and higher energy consumption. A lot of work has been done to mitigate costly write operations on STT-RAM [2], [3], [4]. Recently, researchers propose to improve write speed and write energy of STT-RAM by relaxing the non-volatility property [5]. Table 1 shows an example of different designs of STT-RAM cells [6]. As the retention time decreases, the write latency and write energy consumption are reduced. However, the reduced retention time may not be sufficient to retain long living data in cache blocks. Refresh schemes are proposed to avoid data losses [6], [7]. Refresh operations consume additional energy and overhead. This paper proposes a compilation based approach to significantly reduce the number of refresh operations in volatile STT-RAM caches.

- *Q. Li is with the State Key Laboratory of Software Engineering and the School of Computer, Wuhan University, Wuhan, China. E-mail: qali@whu.edu.cn.*
- *Y. He is with the School of Computer, Wuhan University, Wuhan, China. E-mail: yxhe@whu.edu.cn.*
- *J. Li is with the School of Computer and Information, Hefei University of Technology, Hefei, China. E-mail: jianhual@mail.ustc.edu.cn.*
- *L. Shi is with the School of Computer Science, Chongqing University, Chongqing, China. E-mail: shiliang@cqu.edu.cn.*
- *Y. Chen is with the Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburg, PA 15260. E-mail: yic52@pitt.edu.*
- *C.J. Xue is with the Department of Computer Science, City University of Hong Kong, Hong Kong. E-mail: jasonxue@cityu.edu.hk.*

Manuscript received 4 July 2013; revised 26 Nov. 2013; accepted 11 Dec. 2013. Date of publication 25 Sept. 2014; date of current version 10 July 2015. Recommended for acceptance by J. Xue.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2360527

There are two kinds of operations that can refresh the lifespan of cache blocks. On a write access to a cache block, or when data is loaded from the main memory to a cache block, the block's lifespan is reset to the new-born state, and the data in this block can be stored for another retention period. These kind of operations, triggered by the program behaviour, are called *passive refresh* in this paper. A *passive refresh* cannot guarantee the validity of a data block for the next usage, so a refresh scheme is indispensable. Refresh schemes usually employ counters to track the lifespan of data blocks and provide refresh operations on demand. These kind of operations are called *active refresh* in this paper. An *active refresh* operation involves two operations: loading data from a cache block into a buffer, and storing data back into the same cache block. These operations consume additional energy. Both a *passive refresh* operation and an *active refresh* operation can refresh the whole cache block. Therefore, if there is a refresh operation (either active or passive) within a retention period, no data losses will occur. In this paper, we propose to re-arrange the data layout with the purpose of changing the memory access sequence. As a result, the behaviour of *passive refresh* will be changed such that the total number of required *active refresh* operations is minimized. To deal with data blocks in which the interval between two consecutive accesses are very long, we also propose a refresh scheme which refreshes a data block no more than a predetermined number of times. Reducing the number of refresh operations leads to a reduction of energy consumption. Experimental results show that the proposed methods can reduce the number of refresh operations by 84.2 percent, and reduce the dynamic energy consumption by 38.0 percent on average for volatile STT-RAM caches. This paper makes the following contributions:

- Proposes the problem of *active refresh* minimization in volatile STT-RAM cache by re-arranging the data layout at compilation time;
- Proposes an integer linear programming (ILP) solution for the *active refresh* minimization problem;

TABLE 1
Parameters in Different STT-RAM Cell Designs [6]

	Design 1	Design 2	Design 3
Cell size (F^2)	23	22	27.3
MTJ sw time (ns)	10	5	1.5
Retention Time	4.27 yr	3.24 s	26.5 μs
Write Latency (ns)	10.378	5.370	1.500
Write Dyn. Eng(nJ)	0.958	0.466	0.187

- Proposes a heuristic algorithm to solve the *active refresh* minimization problem;
- Proposes an *N-refresh* scheme to handle data blocks in which the interval between two consecutive accesses are very long;
- Conducts a large set of experiments to evaluate the proposed data layout methods with different refresh schemes.

The rest of this paper is organized as follows. Section 2 discusses several refresh schemes and the motivation. Section 3 describes the problem of data layout for *active refresh* minimization. Section 4 proposes an integer linear programming solution to the *active refresh* minimization problem. Section 5 proposes a heuristic algorithm for *active refresh* minimization. Section 6 presents the experimental setup and the experimental results. Section 7 presents the sensitivity analysis of the proposed methods. The related work and conclusion are presented in Sections 8 and 9 respectively.

2 Refresh Schemes and Motivation

In this section, we first present several refresh schemes, and then present a motivation example to illustrate the potential of *active refresh* reduction through re-arranging the data layout.

2.1 Refresh Schemes

Several refresh schemes have been proposed for volatile STT-RAM designs. Three schemes will be evaluated in the experiments: *full-refresh*, [6], *dirty-refresh*, [7], and *N-refresh* schemes. The *N-refresh* scheme is proposed in this paper.

Full-refresh. Sun et al. in [6] present a refresh scheme to refresh cache blocks asynchronously by tracking the lifespan of each cache block. In this scheme, a counter is attached to each data block, and three types of actions are needed. First, the STT-RAM cells retention time is divided into multiple checking periods, and a global clock is used to maintain the count-down to a checking period. Second, each counter is incremented by one when a checking period elapses. When a counter reaches a predetermined value, the corresponding data block is refreshed. Third, on a write access to a data block or when a data is loaded into a cache block, its corresponding counter is reset to zero. This scheme refreshes all valid data blocks. It is called *full-refresh* scheme in this paper. Table 2 shows that this scheme incurs frequent *active refresh* operations. The architecture parameters that we used for evaluation in this paper are shown in Table 6. On average, there are about 25 *active refresh* operations per hundred memory accesses. These refresh operations lead to significant overload.

TABLE 2
Frequent Refresh under the *Full-Refresh* Scheme

	Instruction Count	Load	Store	Refresh
adpcm	33,508	11,762	7,051	2,453
bcnt	8,543	2,325	1,324	1,058
blit	30,513	4,314	3,328	4,050
crc	16,636	3,705	1,785	2,170
engine	293,053	55,697	49,324	11,131
fir	16,649	3,110	1,766	1,417
g3fax	637,203	81,579	84,278	32,296
pocsag	31,735	5,562	4,856	2,116
qurt	9,202	2,356	1,372	1,083
ucbqsort	310,653	113,781	36,625	14,409

Dirty-refresh. Jog et al. in [7] present a scheme similar to the *full-refresh* scheme. This scheme only refreshes dirty blocks, and discards clean blocks when they are untouched for a retention period. It is called *dirty-refresh* scheme in this paper. Compared to the *full-refresh* scheme, this scheme reduces the number of *active refresh* operations, but also decreases the cache hit ratio.

N-refresh. There are some data blocks in which the intervals between two consecutive writes are very long. An extreme case is the so-called dead blocks which will never be used after a write. Fig. 1 shows that there are about 5.9 percent intervals longer than *13,250* cycles (a retention period). These intervals require *active refresh* to ensure data correctness, and they are called *long intervals* here. Among these *long intervals*, 25.8 percent are longer than 106,000 cycles (eight retention periods). For these intervals, *full-refresh* and *dirty-refresh* schemes need to refresh many times to sustain their lifespan with little benefit. Furthermore, 34.3 percent of these *long intervals* are *dead intervals*. A *dead interval* is an interval starting with a data write but is never read before the next write. For these *dead intervals*, the refresh operations produce only overhead with no benefit.

Based on this observation, we propose an *N-refresh scheme* under which each data block will be refreshed at most $2^N - 1$ times. Therefore, a block untouched for 2^N retention periods will be automatically invalidated and its content will be written back to lower level memory hierarchy. To implement the *N-refresh* scheme, we only need to add N bits to the counter attached to each data block.

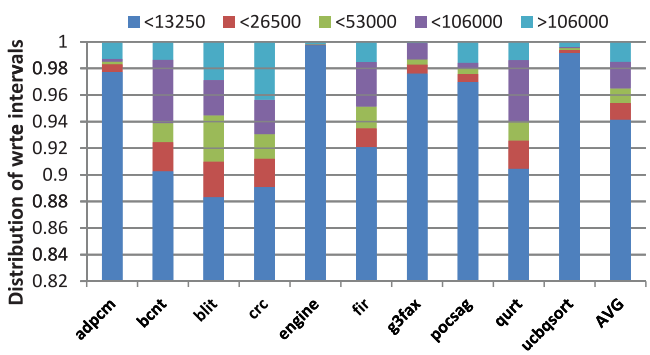


Fig. 1. Distribution of intervals between adjacent memory stores onto the same cache block in clock cycles. The evaluation results are based on a cache with 32-byte block size and 32-bit memory width. More parameters are shown in Table 6.

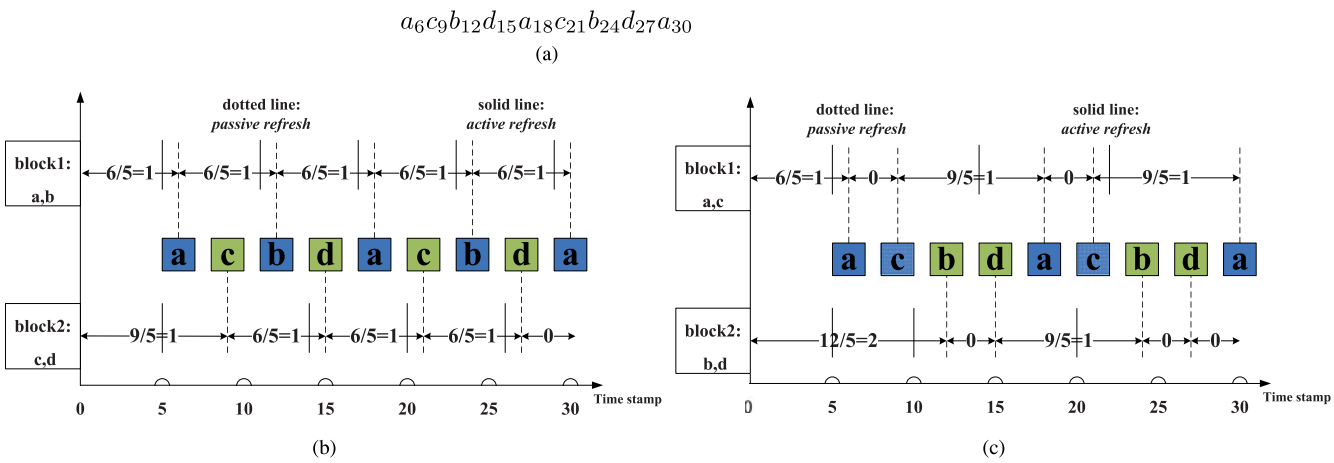


Fig. 2. A motivation example with 5 *ms* retention period. A solid vertical line indicates an *active refresh* operation, and a dotted vertical line indicates a *passive refresh* operation. (a) The data write trace. The subscripts indicate the timestamps for the corresponding writes. (b) The first allocation, $\{a, b\}$ in $block_1$ and $\{c, d\}$ in $block_2$, needs nine *active refresh* operations. (c) The second allocation, $\{a, c\}$ in $block_1$ and $\{b, d\}$ in $block_2$, needs six *active refresh* operations.

Compared to the *full-refresh* scheme, this scheme reduces the number of *active refresh* operations with a small decrease in the cache hit ratio.

2.2 Motivation Example

An example is presented in this section to illustrate the potential of reducing the number of *active refresh* operations through re-arranging the data layout. Assume that there are four data objects (*a*, *b*, *c*, and *d*) and two memory blocks. Each memory block can hold two data objects. The retention time of volatile STT-RAM cache is 5 *ms*. Given a data access trace, a data write trace can be extracted by recording only write accesses, as shown in Fig. 2a. The subscripts indicate the timestamps for the corresponding writes. Two allocations are considered. With the first allocation as illustrated in Fig. 2b, nine *active refresh* operations are needed. However, with the second allocation as illustrated in Fig. 2c, more blocks are passively refreshed within each 5 *ms* period due to write accesses. As a result, only six *active refresh* operations are needed. Based on this observation, this paper proposes to minimize the number of *active refresh* operations for volatile STT-RAM caches by re-arranging the data layout at compilation time.

3 Problem Description

Given a data write trace, either through profiling or static analysis, the goal of this paper is to find an allocation function to allocate data object into memory blocks such that the number of required *active refresh* operations is minimized. Note that as a common practice, data objects are allocated into three disjoint areas, the stack, global, and heap areas. Therefore, data layout should be conducted separately for each area. The notations used in this paper are listed in Table 3. Given a data write trace $DTrace = < dw_1, dw_2, ... >$, the target problem can be stated as follows:

1) The data write trace *DTrace* is transformed into a memory block write trace $BTrace =< bw_1, bw_2, \dots >$, where bw_i and dw_i satisfy: $bw_i = alloc(dw_i)$. If a data object d is allocated to memory block b, then each write to d in *DTrace* will result in a write to b in *BTrace*.
2) The *BTrace* is split into a set of sub-traces. Each sub-trace consists of writes to the same memory block.

   For the example in Fig. 2a, two sub-traces $a_6b_{12}a_{18}b_{24}a_{30}$ and $c_9d_{15}c_{21}d_{27}$ can be obtained for $block_1$ and $block_2$, respectively.

TABLE 3
Notations for the Target Problem

Notation	Detail
$D = \{d_1, \dots, d_n\}$	The set of data objects.
$SizeD_i$	The size of d_i in bytes.
$B = \{b_1, \dots, b_m\}$	The set of memory blocks. A memory block is the basic unit for cache loading and storing operations.
$SizeB$	The size of a memory block in bytes.
$DTrace = <dw_1, dw_2, \dots, dw_i, \dots>$	The data write trace, a sub-trace of the data trace with only writes recorded.
DO_i	The target data object of data write dw_i.
TS_i	The time stamp for data write dw_i.
TS_{start}	The time stamp when the program starts.
TS_{end}	The time stamp when the program ends.
$SizeW_i$	The size of dw_i. By preprocessing, if dw_i is the first occurrence of DO_i, $SizeW_i$ is $SizeD_{DO_i}$; otherwise, it is zero.
$BTrace = <bw_1, bw_2, \dots, bw_i, \dots>$	The memory block write trace.
T	The retention time of STT-RAM cells.
$alloc : D \rightarrow B$	The allocation function to allocate data objects into memory blocks.

3) For each sub-trace $trace_{b_j} = <bw_1^j, bw_2^j, \ldots>$ for block b_j, the number of required *active refresh* operations can be computed using Equation (1). This is because that, assuming the retention time is T, the number of required *active refresh* operations within each time interval inv is $\lfloor inv/T \rfloor$.

$$\begin{aligned} nAR_{b_j} = & \sum_{i=1}^{|trace_{b_j}|-1} \lfloor (TS_{bw_{i+1}^j} - TS_{bw_i^j})/T \rfloor \\ & + \lfloor (TS_{bw_1^j} - TS_{start})/T \rfloor \\ & + \lfloor (TS_{end} - TS_{bw_{|trace_{b_j}|}^j})/T \rfloor. \end{aligned} \quad (1)$$

For simplicity, in this paper, we do not model the cache behaviour, such as cache hits/misses and thus the *passive refresh* resulting from cache block loading is ignored. As a result, Equation (1) is employed for problems under both *full-refresh* and *dirty-refresh* schemes. For problems under *N-refresh* scheme, this equation is not accurate. This is because that, assuming the retention time is T, the number of required *active refresh* operations within each time interval inv should not be greater than $2^N - 1$. Therefore, the number of required *active refresh* operations is $\min\{\lfloor inv/T \rfloor, 2^N - 1\}$, and Equation (1) should be changed accordingly.

4) The goal is to find an allocation function $alloc$ to minimize:

$$\sum_{j}^{|N|} nAR_{b_j}. \quad (2)$$

Note that any re-arrangement of data layout will affect the program locality. The impact of program locality is not integrated directly into this formulation. This limitation will be discussed in Section 6. In addition, we do not re-arrange data layout of the heap area, since the heap area is rarely used in embedded applications and it is difficult to manage heap area at compilation time. Data objects with size greater than the cache block size will be allocated using the default method.

4 ILP Formulation

In this section, an integer linear programming ormulation is presented for the target problem. ILP is a mathematical method to achieve the best outcome (such as maximum profit or lowest cost) in a given model, under a list of constraints represented as linear relationships and a specific constraint that the solution should be modelled within the integer domain. Once a problem is formulated as ILP, advanced and sophisticated mathematical techniques can be employed to search the optimal results by simply applying commercial ILP solvers. ILP is NP-hard in general, and ILP solvers commonly employ implicit enumeration techniques to obtain the optimal results. Although ILP solvers often consume lots of computation resources, they can obtain the optimal results for relatively small problems within an acceptable time frame. Furthermore, ILP solutions can also help to evaluate the effectiveness of other efficient but suboptimal algorithms.

In this section, we first present the ILP formulation for the allocation problem under *full-refresh* and *dirty-refresh* schemes. Then, an example is shown to explain this formulation. Finally, we discuss the formulation for the problem under *N-refresh* scheme.

We use x_i^j to indicate whether the target data object of dw_i is allocated into memory block b_j, as defined in Equation (3).

$$x_i^j = \begin{cases} 1, & \text{if } dw_i \text{ isallocatedintomemoryblock } b_j; \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

If the target data object of a data write dw_i is allocated into memory block b_j, and dw_i occurs at time stamp TS_i, then a memory write occurs in b_j at TS_i; otherwise, dw_i occurs in other memory blocks, and at TS_i no memory write occurs in b_j. We use pts_i^j to denote the pseudo time stamp when data write dw_i occurs in b_j, as defined in Equation (4). This set of pseudo time stamps are used to simplify the ILP formulation.

$$pts_i^j = \begin{cases} TS_{start}, & \text{if } i = 0; \\ TS_{end}, & \text{if } i = |DTrace| + 1; \\ TS_i, & \text{if } 1 \le i \le |DTrace| \text{ and } dw_i \text{ is in } b_j; \\ pts_{i-1}^j, & \text{otherwise.} \end{cases} \quad (4)$$

To comply with the definition in Equation (4), we use Equation (5) to constrain the value of pts_i^j. The first line states that the first pseudo time stamp for each block is TS_{start}. The second line states that the last pseudo time stamp for each block is TS_{end}. The following three lines constrain the other pseudo time stamps. If dw_i actually occurs in b_j, $x_i^j = 1$. Then, the fifth line leads to $pts_i^j \ge TS_i$. Together with the third line, it constrains that the value of pts_i^j can only be TS_i. If dw_i doesn't occur in b_j, $x_i^j = 0$. Then, the fourth line leads to $pts_i^j \le pts_{i-1}^j$. Together with the third line, it constrains that the value of pts_i^j can only be pts_{i-1}^j.

$$\begin{aligned} & \forall j \in \{1, 2, \ldots, |B|\}, pts_0^j = TS_{start} \\ & \forall j \in \{1, 2, \ldots, |B|\}, pts_{|DTrace|+1}^j = TS_{end} \\ & \forall i \in \{1, 2, \ldots, |DTrace|\}, pts_{i-1}^j \le pts_i^j \le TS_i \\ & \forall i \in \{1, 2, \ldots, |DTrace|\}, pts_i^j \le pts_{i-1}^j + x_i^j * TS_i \\ & \forall i \in \{1, 2, \ldots, |DTrace|\}, pts_i^j \ge TS_i + (x_i^j - 1) * TS_i. \end{aligned} \quad (5)$$

For each memory block, there must be an *active refresh* or a *passive refresh* (a memory write) within the retention time T, to avoid data losses. If the interval between two consecutive writes in the block is equal to or longer than T, extra *active refresh* is needed. The number of required *active refresh* in memory block b_j, between consecutive data writes dw_i and dw_{i+1} from the whole trace, can be computed using Equation (6). As indicated in this Equation, if dw_{i+1} does not occur in b_j, pts_{i+1}^j equals pts_i^j, and no *active refresh* is needed.

$$nAR_i^j = \lfloor (pts_{i+1}^j - pts_i^j)/T \rfloor. \quad (6)$$

However, Equation (6) is not linear. Instead, we use the following linear Equation (7).

TABLE 4
An Example for ILP Illustration

	i	0	1	2	3	4	5	6	7	8	9	10	$cost^j$
	$DTrace$	*start*	a	c	b	d	a	c	b	d	a	*end*	
	TS_i	0	6	9	12	15	18	21	24	27	30	31	
$block_1$	sub-trace$_1$	*start*	a	c	—	—	a	c	—	—	a	*end*	3
	pts_i^1	0	6	9	9	9	18	21	21	21	30	31	
{a,c}	nAR_i^1	0	$\lfloor\frac{6}{5}\rfloor=1$	0	0	0	$\lfloor\frac{9}{5}\rfloor=1$	0	0	0	$\lfloor\frac{9}{5}\rfloor=1$	0	
$block_2$	sub-trace$_2$	*start*	—	—	b	d	—	—	b	d	—	*end*	3
	pts_i^2	0	0	0	12	15	15	15	24	27	27	31	
{b,d}	nAR_i^2	0	0	0	$\lfloor\frac{12}{5}\rfloor=2$	0	1	0	$\lfloor\frac{9}{5}\rfloor=1$	0	1	0	

$$\begin{aligned}&\forall j \in \{1,2,\ldots,|B|\}, nAR_0^j = 0.\\ &\forall i \in \{1,2,\ldots,|DTrace|+1\}, \left(pts_i^j - pts_{i-1}^j\right) - T + 1\\ &\qquad\qquad \leq nAR_i^j \cdot T \leq pts_i^j - pts_{i-1}^j.\end{aligned} \tag{7}$$

Then, the cost (the number of required *active refresh*) resulting from memory block b_j can be described using Equation (8).

$$cost^j = \sum_{i=1}^{|DTrace|+1} nAR_i^j. \tag{8}$$

The objective is to minimize the total cost depicted in Equation (9).

$$\sum_{j=1}^{|B|} cost^j. \tag{9}$$

There are further constraints to ensure that the allocation is valid. First, each *data write* (from the original trace) can be allocated into only one memory block. This is depicted using Equation (10). Here we assume that the program start point and end point are allocated into every memory block. An example at the end of this section will show that this assumption helps to compute the required *active refresh* before the first touch of each block and after the last touch of each block.

$$\begin{aligned}&\forall i \in \{1,2,\ldots,|DTrace|\}, \sum_{j=1}^{|B|} x_i^j = 1\\ &\forall j \in \{1,2,\ldots,|B|\}, x_0^j = 1\\ &\forall j \in \{1,2,\ldots,|B|\}, x_{|DTrace|+1}^j = 1.\end{aligned} \tag{10}$$

Second, each *data object* can only be allocated into one memory block, as described in Equation (11). Therefore, all *data writes* that target at the same *data object* must be allocated into the same memory block, as depicted using Equation (11).

$$\begin{aligned}\forall i_1, i_2 \in \{1,2,\ldots,|DTrace|\}, DO_{i_1} = DO_{i_2},\\ \forall j \in \{1,2,\ldots,|B|\},\\ x_{i_1}^j = x_{i_2}^j.\end{aligned} \tag{11}$$

Furthermore, the total size of data objects allocated into the same memory block should not be greater than the size of one memory block, as described in Equation (12). Note that if a data write dw_i is not the first occurrence of a data object d_j in the trace, the size of dw_i is set to zero during preprocessing.

$$\forall j \in \{1,2,\ldots,|B|\}, \sum_{i=1}^{|DTrace|} x_i^j \cdot SizeW_i \leq SizeB. \tag{12}$$

In total, there are three sets of important variables, x_i^j, pts_i^j, and nAR_i^j. Among these variables, x_i^j should be binary values, pts_i^j and nAR_i^j should be integer values.

Here we use the previous example in Fig. 2c to illustrate the intrinsic meanings of the above ILP notations. As shown in Table 4, the $DTrace$ is $<a;c;b;d;a;c;b;d;a;>$. With $\{a,c\}$ allocated in $block_1$, the *sub-trace*$_1$ is $<a;c;a;c;a>$. Assume that TS_{start} is 0, and TS_{end} is 31. The value of pts_i^j can be computed using Equation (5), as shown in the fifth and eighth rows of Table 4. Similarly, the value of $cost^j$ can be computed using Equation (8), as shown in the last column of Table 4.

To adapt this ILP formulation for the problem under *N-refresh* scheme, we only need to change Equation (7) into Equation (13), since under *N-refresh* scheme, each memory block can be consecutively refreshed at most $2^N - 1$ times.

$$\begin{aligned}&\forall j \in \{1,2,\ldots,|B|\}, nAR_0^j = 0\\ &\forall i \in \{1,2,\ldots,|DTrace|+1\}, nAR_i^j \cdot T \leq pts_i^j - pts_{i-1}^j\\ &\forall i \in \{1,2,\ldots,|DTrace|+1\}, nAR_i^j \leq 2^N - 1.\end{aligned} \tag{13}$$

5 A Heuristic Algorithm

As ILP is not scalable for large problem sets, we propose an efficient and effective heuristic algorithm in this paper. We consider only pairwise information between data objects during the data layout process to simplify the target problem. For each pair of data objects, d_i and d_j, we extract a sub-trace by removing all other data objects from the data write trace. We denote this sub-trace as $trace_{i,j} = <dw_1, dw_2, \ldots>$. Then, the cost of assigning d_i and d_j into the same block is approximated as:

$$cost_{i,j} = \sum_{k=1}^{|trace_{i,j}|-1} \lfloor (TS_{dw_{k+1}} - TS_{dw_k})/T \rfloor. \tag{14}$$

If d_i and d_j are assigned into different blocks, this cost is zero. This simplified problem can be modelled as a quadratic

assignment problem (QAP) for which a graph representation can be employed. In this graph, a vertex represents a data object, the weight of an edge connecting two vertices represents the cost of assigning the two related data objects into the same block. Now the target problem is transformed to partitioning the graph into its vertex-induced sub-graphs, where each sub-graph corresponds to a block, such that the total cost of all sub-graphs is minimized.

Since QAP is NP-hard in general [8], a heuristic algorithm is proposed in this paper. This algorithm consists of two steps: graph partitioning and data layout finalization. The proposed algorithm re-arranges data layout for global area and stack area separately and in the similar fashion. Therefore, only data layout for stack area is discussed in detail in the rest of this paper.

5.1 Graph Partitioning

The graph partitioning for stack data layout is conducted separately for each function, and is illustrated in Algorithm 5.1. First, it builds a list of empty memory blocks to be allocated. Then, it allocates data objects into blocks, by iteratively retrieving the edge with the smallest weight and allocating the related two data objects into the same block. After each allocation, the graph is updated accordingly.

Algorithm 5.1. Graph Partitioning.

Input:
graph: the graph representation;
blocks: an empty list of memory blocks;
nStackSize: the stack size for a 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: allocate data into the list of memory blocks*
7: **while** *graph* is not empty **do**
8: retrieve an edge $e(v_1, v_2)$ from *graph* with the smallest weight;
9: // *try to allocate* v_1, v_2 *into the same block*
10: **if** both v_1 and v_2 are unallocated **then**
11: allocate v_1 and v_2 into the same block;
12: merge vertex v_1 and v_2;
13: **else if** only one object is unallocated **then**
14: allocate it into the block holding the other object;
15: merge vertex v_1 and v_2;
16: **else**
17: delete edge $e(v_1, v_2)$ from the graph;
18: **end if**
19: **end while**
20: // *Step 3: allocate the other data using the default method*
21: allocate the unallocated data using the default method;
22: **return** *blocks*;

An example is illustrated in Fig. 3 to show the graph partitioning process. In the proposed algorithm, the weighted graph is a complete graph. It is assumed that all data objects are of the same size, and that each cache block (or memory block) can hold three objects. Since there are

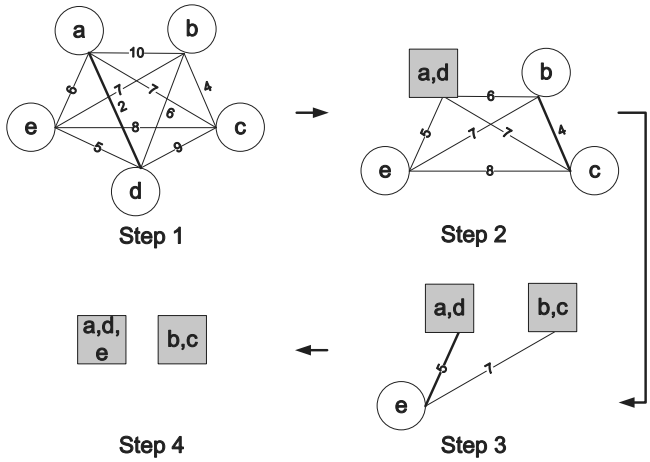


Fig. 3. Graph partitioning.

five objects, the memory blocks list is initialized with two (5/3) empty blocks. In the first step, the edge connected by node a and node d has the smallest weight, 2. So node a and node d are allocated into the first memory block and they are merged into one node (a, d). During this process, the edges connected by a and d are merged. For example, the edges (a, e) and (d, e) are merged into (a, e), and the weight is updated (5=$\min\{5, 6\}$). The final allocation is shown in the fourth step.

Let's discuss the complexity of the graph partitioning process. Assume that the number of global objects is N_g and the number of stack objects in function f_i is N_{f_i}. We denote $\max\{N_g, N_{f_1}, N_{f_2}, \ldots\}$ as M. Then the number of nodes and edges in a graph is no more than M and M^2 respectively. The kernel code is the loop starting at line 7. In this loop, since each iteration can erase an edge from the graph, the total number of iterations is bounded by M^2. The time cost of each iteration is limited by $O(M)$. Therefore, the timing complexity for Algorithm 5.1 is $O(M^3)$. For a compilation optimization pass, this time cost is acceptable.

5.2 Data Layout Finalization

After the graph partition process, a list of memory blocks containing data objects are obtained. The offset of each data object internal to its memory block is also obtained. Now, we will conduct the layout finalization work for the stack. This work assigns each data object into the stack address space according to its memory block and offset. The finalization work involves two processes, the alignment process and the mapping process.

The alignment process aims to align the stack base pointer of each function with the cache block size. Two tasks are carried out for this process. 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 block size. Second, the stack size of each function is expanded to be a multiple of the cache block size.

The mapping process aims to assign data objects to addresses. After the alignment work, we can finalize the stack layout by mapping the memory blocks into the stack directly. Assume that the cache block size is *C*, a data object d belongs to the nth memory block, and the offset is *off*.

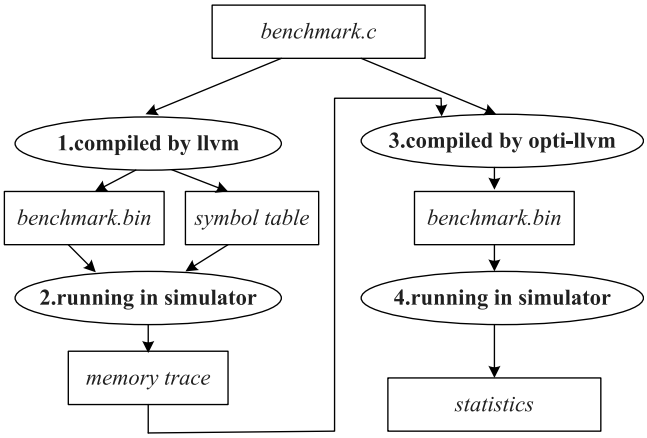


Fig. 4. The experimental setup.

During the mapping process, we simply allocate d to $ESP+n*C+off$. With this mapping process, the objects previously allocated into the same memory block using Algorithm 5.1 will also be loaded into the same cache block. Note that some special contents belonging to the stack could not be re-arranged freely. Therefore, the mapping of memory blocks should be conducted after the space for all special contents have already been reserved.

6 Experiments

In this section, we first introduce the experimental methodology, and then present the experimental results of the proposed methods.

6.1 Experimental Methodology

As stated in Section 2.1, there are three kinds of STT-RAM refresh schemes: *full-refresh*, *dirty-refresh*, and *N-refresh* schemes. There are also three data layout methods discussed in this paper, default data layout, heuristics based data layout (Section 5), and ILP based data layout (Section 4). The refresh schemes and the data layout methods are orthogonal. The combination of refresh schemes and data layout methods constitutes nine methods as shown in Table 5. All these nine methods are evaluated in this paper. Here we choose N as 1 for the *N-refresh* scheme.

To evaluate these methods, we implement a PIN-based [9] cache simulator. This simulator implements all three refresh schemes in Table 5 and targets at embedded systems where single core with one-level cache is widely used. The architecture parameters shown in Table 6 are based on TI DM3x Video SOC [10]. The Powerstone benchmark suite [11] is evaluated in the experiments which contains typical benchmarks for embedded systems.

The experimental setup is shown in Fig. 4, consisting of four steps. First, a benchmark is compiled using the original LLVM [12]. The output includes the binary and the symbol table information for this benchmark. Second, by running the binary in the PIN-based cache simulator, the memory trace with timestamps is obtained. With the symbol table information, a memory trace is transformed into a data write trace. Third, with this data write trace, the benchmark is re-compiled by LLVM. This time, the data layout of stack and global objects is re-arranged using one of the data layout methods shown in Table 5. The ILP based methods are implemented using a commercial ILP solver, LINGO [13]. Fourth, by running the binaries with the re-arranged data layout in the same cache simulator, the statistics are collected. For all methods, we evaluate the impact on both dynamic energy consumption and performance.

TABLE 5
Evaluated Methods

	full-refresh	*dirty-refresh*	*N-refresh*
Default data layout	FR	DR	NR
Heuristics data layout	FR-DL	DR-DL	NR-DL
ILP based layout	FR-ILP	DR-ILP	NR-ILP

TABLE 6
Architecture Parameters

Parameter	Value
processor	single core, in order execution, 500 MHz
data cache	16 KB, 32 B cache block size, LRU, 4-way write allocation, write back a small SRAM buffer used for refresh read/write latency: 1/1 cycles retention time: 13250 cycles (26.5μs) read/write dynamic energy: 0.035/0.187 nJ buffer read/write dynamic energy: 0.075/0.059 nJ dynamic energy of *active refresh*: 0.356 nJ
main memory	300 cycles latency

6.2 Experimental Results

In this section, we first evaluate the proposed methods and present the results of dynamic energy and performance respectively. The reduction of dynamic energy is mainly due to the reduction of *active refresh*. The improvement of performance is mainly due to the increase of cache hit ratio. Note that the ILP solver can obtain the optimal results for only four out of ten benchmarks. We then evaluate partial traces for the other six benchmarks.

6.2.1 *Impact on* Active Refresh *and Dynamic Energy*

We first discuss the impact on *active refresh*. Fig. 5a shows the number of *active refresh* operations of the nine methods for different benchmarks. All results are normalized to the FR method. It is found that, compared to the default data layout, the proposed heuristic method can further reduce the number of *active refresh* by 12.4, 15.7, and 2.0 percent under the *full-refresh*, *dirty-refresh*, and *N-refresh* schemes, respectively. It also shows that, for the four benchmarks of which the ILP method can obtain the results, the ILP method can reduce more *active refresh* than the heuristic method under all refresh schemes. This is because the ILP method pursues the optimal results.

In addition, compared to the *full-refresh* scheme, with the default data layout, the *dirty-refresh* and *N-refresh* schemes can reduce the number of *active refresh* by 42.7 and 82.4 percent respectively, on average. Combining the effects of both refresh schemes and data layout methods, compared to the

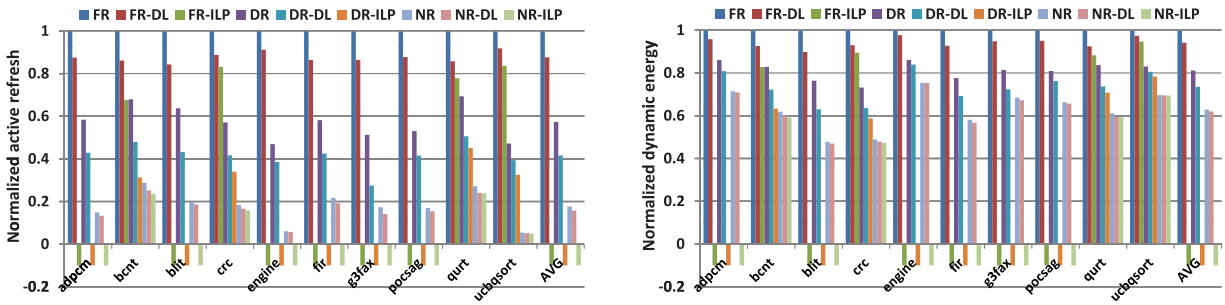


Fig. 5. Impact on energy efficiency. All results are normalized to the FR method. A negative value indicates that the ILP method cannot obtain the result for the corresponding benchmark within three days. (a) Comparison of *active refresh* operations. (b) Comparison of dynamic energy consumption on data cache.

FR method, the DR-DL, and NR-DL methods can reduce the number of *active refresh* by 58.4 and 84.2 percent respectively. It also shows that, on average, *N-refresh* scheme reduces more *active refresh* than *dirty-refresh* scheme. However, as the following results shows, *dirty-refresh* has better performance compared with *N-refresh*.

Next we present the impact on dynamic energy consumption. Fig. 5b shows the dynamic energy consumption of the nine methods. It is found that, compared to the default data layout, the proposed data layout methods can consistently reduce the dynamic energy consumption for all refresh schemes. This is consistent to the impact on refresh operations. Compared to the FR method, on average, the DR-DL and NR-DL methods can reduce the dynamic energy consumption in data cache by 26.4 and 38.0 percent, respectively. It also shows that, for the four benchmarks of which the ILP method can obtain results, compared to the default data layout, the heuristic (ILP) data layout can improve energy efficiency by 6.1 percent (11.2 percent), 8.2 percent (12.9 percent), and 1.2 percent (1.6 percent), in *full-refresh*, *dirty-refresh*, and *N-refresh* schemes respectively.

6.2.2 Impact on Performance

In this section, we present the impact on performance in terms of cache hit ratio. Consider the impact of different data layout methods. As stated in Section 3, the impact of cache hit ratio is not integrated into the problem model directly. However, since the goal of the problem is to reduce intervals between consecutive cache writes in the same cache blocks, a solution to this problem helps to reduce the reuse instance for each cache block, and naturally benefits the program locality. This is confirmed by the experiments. As shown in Fig. 6b, compared to the default data layout, both proposed data layout methods can improve the cache hit ratio slightly under the same refresh schemes for all benchmarks. In addition, the ILP method works slightly better than the heuristic method.

Now consider the impact of different refresh schemes. Compared to the *full-refresh* scheme, both the *dirty-refresh* and *N-refresh* schemes decrease cache hit ratio with the same data layout methods. The *dirty-refresh* scheme only refreshes dirty blocks and thus content of non-dirty blocks are lost after untouched for the retention time. The *N-refresh* scheme only refreshes for at most once, and thus content of blocks are lost after untouched for two retention periods. Data losses in cache blocks lead to the decrement in cache hit ratio. Combining the impact of both data layout methods and refresh schemes, compared to the FR method, the DR-DL and NR-DL methods slightly decrease the cache hit ratio by 2.3 and 1.6 percent respectively.

The impact on cache hit ratio affects the performance, as shown in Fig. 6a. Compared to the default data layout, the proposed data layout can always reduce the execution cycles with the same refresh schemes. However, compared to the *full-refresh* scheme, both the *dirty-refresh* and *N-refresh* increase the execution cycles with the same data layout methods. This is consistent with the impact on cache hit ratio. Combining the impact of both data layout methods and refresh schemes, the DR-DL method increases the execution cycles by 3.9 percent while the NR-DL method increases the execution cycles by 4.1 percent compared to the FR method.

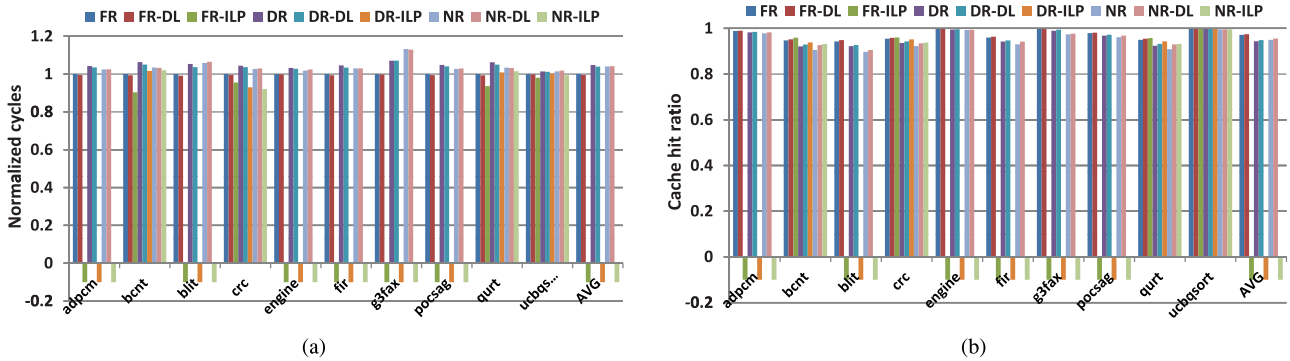


Fig. 6. Impact on performance. A negative value indicates that the ILP method cannot obtain the result for the corresponding benchmark within three days. (a) Comparison of execution cycles on data cache. All results are normalized to the FR method. (b) Comparison of hit ratio on data cache.

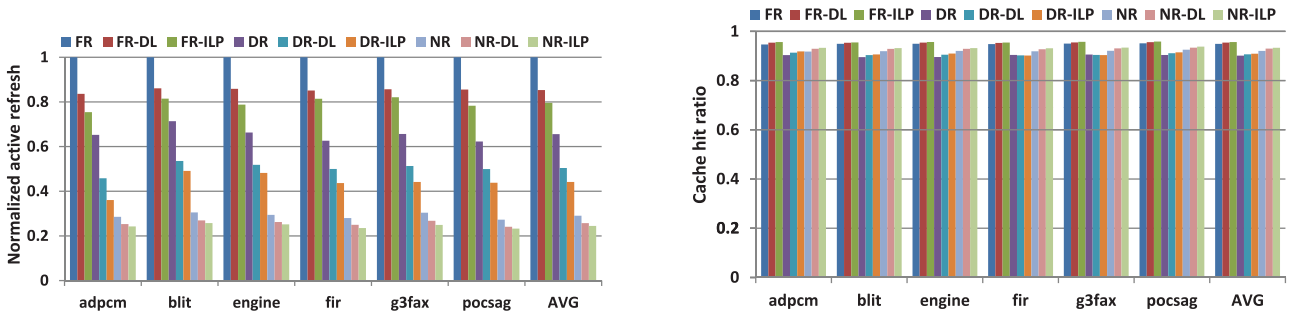


Fig. 7. Evaluation of partial traces. (a) Comparison of *active refresh* operations. Normalized to the FR method. (b) Comparison of cache hit ratio.

6.2.3 Evaluation Based on Partial Traces

Here we evaluate partial traces for six benchmarks, since it is very hard to obtain the ILP optimal results for their traces. We use the first 10, 30, 15, 25, 1, and 25 percent traces for *adpcm*, *blit*, *engine*, *fir*, *g3fax*, and *pocsag*, respectively. We evaluate the proposed methods in measure of dynamic energy and performance respectively. As stated previously, the reduction of dynamic energy is mainly due to the reduction of active refresh, and the improvement of performance is mainly due to the increment of cache hit ratio. Hence, we only present the evaluation results for *active refresh* and cache hit ratio.

Fig. 7a shows the comparison of *active refresh* operations. There are two observations. First, compared to the default data layout method, the proposed heuristics based data layout method can consistently reduce the number of *active refresh* under different refresh schemes. In addition, the proposed ILP based data layout method outperforms the heuristic method for all benchmarks. Second, among all three refresh schemes, both *dirty-refresh* and *N-refresh* schemes can consistently reduce more *active refresh* than the *full-refresh* scheme. It is found that for all partial traces, the *N-refresh* scheme can reduce more *active refresh* than the *dirty-refresh* scheme. It also shows that, the ILP method can reduce more *active refresh* than the heuristic method under all refresh schemes. This is because the ILP method pursues the optimal results.

Fig. 7b shows the comparison of cache hit ratio. There are two observations. First, compared to the default data layout method, the proposed heuristics based data layout method can consistently improve the program locality under different refresh schemes. In addition, the proposed ILP based data layout method outperforms the heuristic method for all benchmarks. Second, compared to the *full-refresh* scheme, both *dirty-refresh* and *N-refresh* schemes degrade the cache hit ratio with the default data layout method. This is because that, the latter two schemes invalidate more cache blocks, which leads to more cache misses. However, together with the proposed data layout methods, the program locality is close to the baseline.

7 Sensitivity Analysis

We have conducted a set of experiments to evaluate the sensitivity of the proposed methods to cache block size, cache size, write latency, retention time, and the value of N for *N-refresh* scheme. For each sensitivity evaluation, we use the parameters shown in Table 6. We evaluate the proposed methods in measure of dynamic energy and performance respectively. We present the evaluation results for *active refresh* and cache hit ratio in this section.

7.1 Sensitivity of Cache Block Size

Fig. 8a shows the *active refresh* operations under different cache block size. There are three observations. First, compared to the default data layout method, the proposed data layout method can consistently reduce the number of *active refresh* under different cache block size. Second, among all three refresh schemes, both *dirty-refresh* and *N-refresh* schemes can consistently reduce more *active refresh* operations than the *full-refresh* scheme. Third, as the cache block size becomes larger, the number of required *active refresh* decreases. This is because that, with the total cache size unchanged, larger cache block size indicates less cache blocks, and less cache blocks indicate less refresh operations.

Fig. 8b shows the cache hit ratio under different cache block size. There are three observations. First, compared to the default data layout method, the proposed data layout method can consistently improve the program locality under different cache block size. Second, compared to the *full-refresh* scheme, both *dirty-refresh* and *N-refresh* schemes degrade the cache hit ratio. This is because that, the latter two schemes invalidate more cache blocks, which leads to more cache misses. Third, as the cache block size becomes larger, the *dirty-refresh* and *N-refresh* schemes have a smaller influence on cache hit ratio. This phenomenon is mainly due to the fact that, with the total cache size unchanged, larger cache block size indicates less cache blocks and thus less cache blocks invalidated by the latter schemes.

7.2 Sensitivity of Total Cache Size

Fig. 9a shows the *active refresh* operations under different cache size. There are four observations. First, compared to the default data layout method, the proposed data layout method can consistently reduce the number of *active refresh* under different cache size. Second, among all three refresh schemes, both *dirty-refresh* and *N-refresh* schemes can consistently reduce more *active refresh* operations than the *full-refresh* scheme. Third, as the cache size becomes larger, the number of required refresh operations increases. This is because that, given the cache block size, a large cache size indicates more cache blocks and thus more refresh operations. Fourth, when the cache size is larger than 32 KB, the number of required *active refresh*

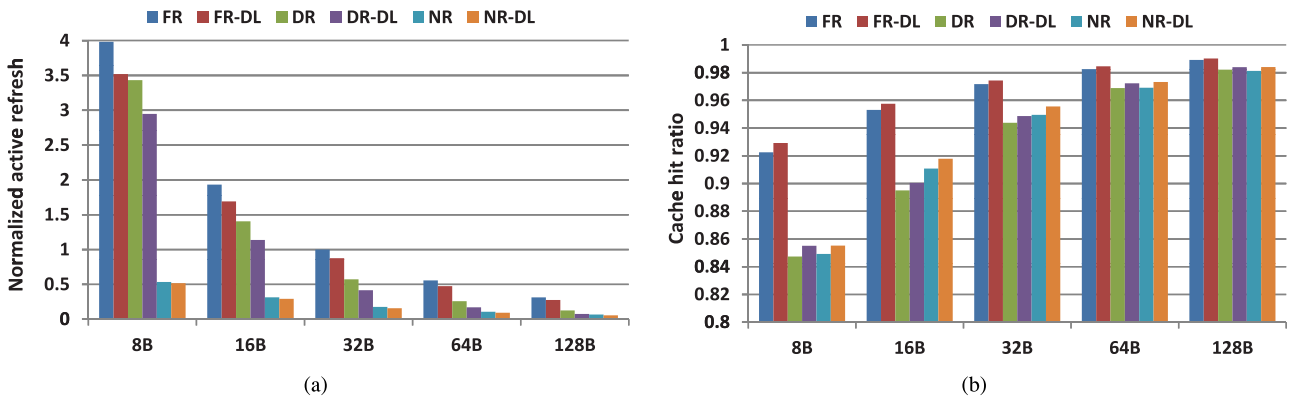


Fig. 8. Sensitivity analysis of cache block size. (a) Comparison of *active refresh* operations. Normalized to the FR method in 32 KB cache size. (b) Comparison of cache hit ratio.

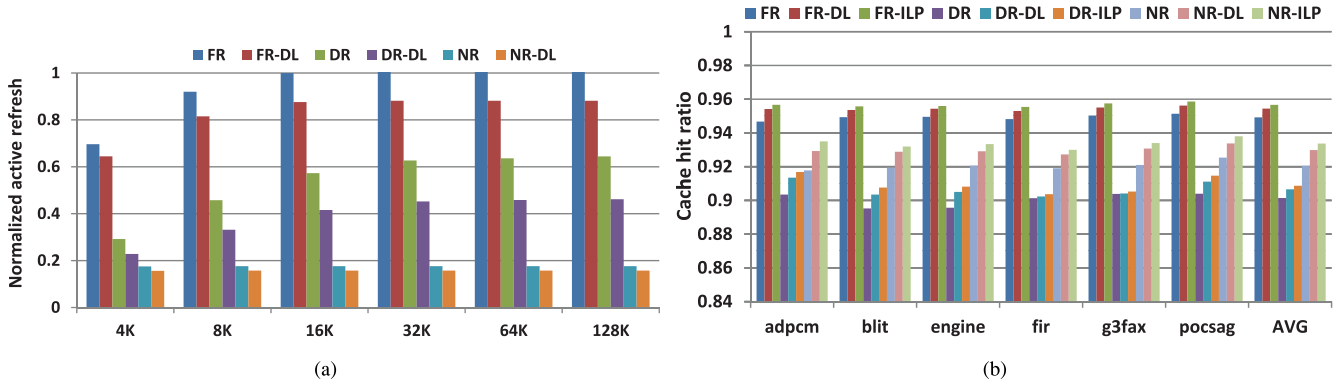


Fig. 9. Sensitivity analysis of total cache size. (a) Comparison of *active refresh* operations. Normalized to the FR method in 64-byte block size. (b) Comparison of cache hit ratio.

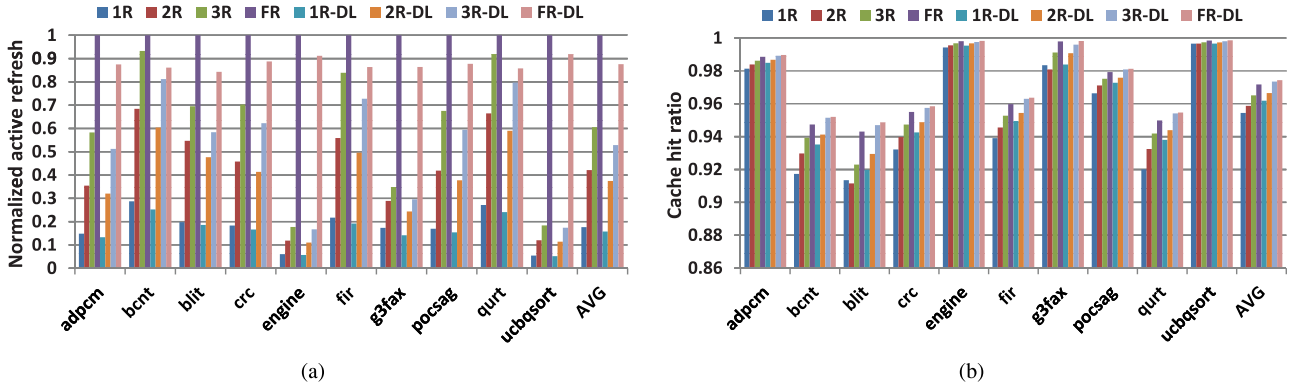


Fig. 10. Sensitivity analysis of the value of *N*. In the figures, nR represents the NR method with the value of *N* is *n*; nR-DL represents the NR-DL method with the value of *N* is *n*. (a) Comparison of *active refresh* operations. Normalized to the FR method. (b) Comparison of cache hit ratio.

operations almost remains unchanged. This phenomenon is mainly due to the fact that, for these benchmarks, a data cache of 32 KB is large enough such that additional cache space is rarely used.

Fig. 9b shows the cache hit ratio under different cache size. There are three observations. First, compared to the default data layout method, the proposed data layout method can consistently improve the program locality under different cache size. Second, when the cache size is larger than 16 KB, the cache hit ratio almost remains unchanged. This phenomenon is mainly due to the fact that, for these benchmarks, a data cache of 32 KB is enough, and additional cache space are rarely used. Third, in general, the *N-refresh* scheme obtains slightly better program locality than the *dirty refresh* scheme.

7.3 Sensitivity of *N* for the *N-Refresh* Scheme

Fig. 10a shows the *active refresh* operations under different values of *N*. There are three observations. First, compared to the default data layout method, the proposed data layout method can consistently reduce the number of *active refresh* under different value of *N*. Second, as the value of *N* becomes larger, the number of required *active refresh* operations increases. This is because that, a larger

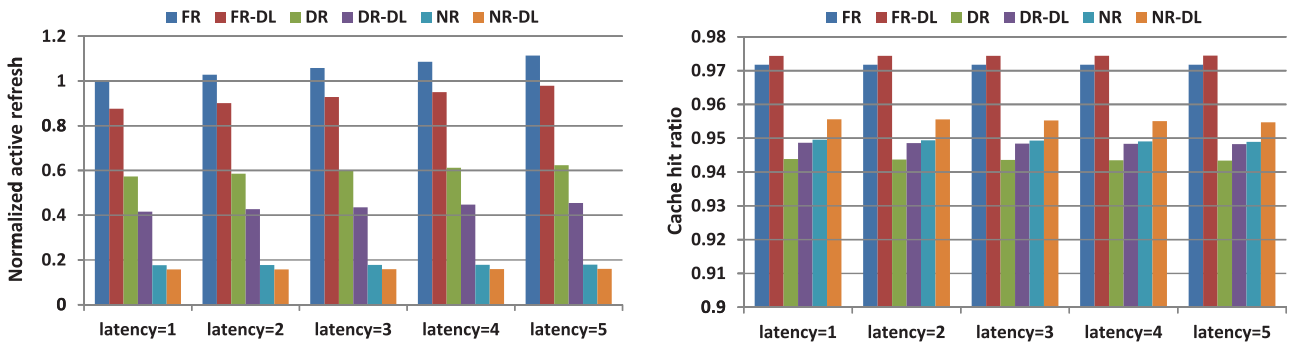


Fig. 11. Sensitivity analysis of write latency. (a) Comparison of *active refresh* operations. Normalized to the FR method with write latency of four cycles. (b) Comparison of cache hit ratio.

value of N indicates more refresh operations for each interval, and thus more refresh operations are conducted. Third, as the value of N becomes larger, the proposed data layout method works better since the optimization space becomes larger.

Fig. 10b shows the cache hit ratio under different values of N. There are two observations. First, compared to the default data layout method, the proposed data layout method can consistently improve the program locality under different value of N. Second, as the value of N becomes larger, the cache hit ratio increases. This because that, a large value of N indicates more refresh operations for each interval, and thus less cache blocks are invalidated.

7.4 Sensitivity of Write Latency

We have conducted the sensitivity analysis of write latency. Note that it is not easy to tune the parameters due to the high complexity of the MTJ model. Therefore, a simulated range of write latency are used to conduct the sensitivity analysis for the proposed methods.

Fig. 11a shows the *active refresh* operations under different write latencies. There are two observations. First, compared to the default data layout method, the proposed data layout method can consistently reduce the number of *active refresh* under different write latencies. Second, as the write latency becomes longer, the number of required *active refresh* operations increases. This is mainly due to the fact that, as the write latency becomes longer, the execution time becomes longer and thus more refresh operations are required.

Fig. 11b shows the cache hit ratio under different write latencies. It is found that, compared to the default data layout method, the proposed data layout method can consistently improve the program locality under different write latencies.

7.5 Sensitivity of Retention Time

We have conducted the sensitivity analysis of retention time. Note that it is not easy to tune the parameters due to the high complexity of the MTJ model. Therefore, a simulated range of retention time are used to conduct the sensitivity analysis for the proposed methods.

Fig. 12a shows the *active refresh* operations under different retention times. There are two observations. First, compared to the default data layout method, the proposed data layout method can consistently reduce the number of *active refresh* under different retention times. Second, as the retention time becomes larger, the number of required *active refresh* operations decreases. In the meantime, the improvement of the proposed data layout method degrades since the optimization space is more limited.

Fig. 12b shows the cache hit ratio under different retention times. There are two observations. First, compared to the default data layout method, the proposed data layout method can consistently improve the program locality under different retention times. Second, as the retention time becomes longer, the *dirty-refresh* and *N-refresh* schemes has less impact on program locality. This is because that, as the retention time becomes larger, the number of cache misses due to these two incomplete refresh schemes decreases.

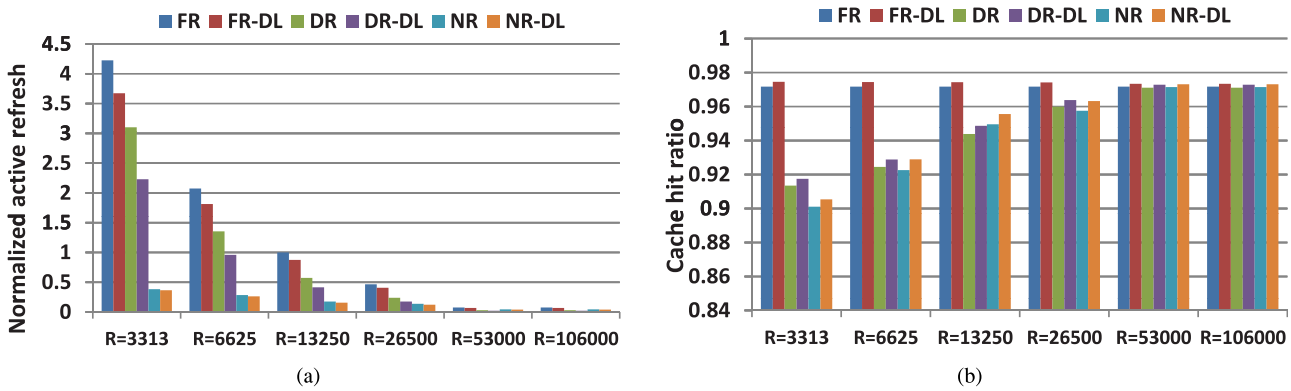


Fig. 12. Sensitivity analysis to different retention times. (a) Natural logarithm of *active refresh* operations, normalized to the FR method with retention time of *53,000* cycles ($26.5\mu s$). (b) Comparison of cache hit ratio.

8 Related Work

In this section, we first discuss prior work that aims to reduce the negative effects of STT-RAM caches. Then, we discuss prior work that aims to reduce the negative effects of refresh.

8.1 Prior Work on STT-RAM Caches

Researchers are motivated to apply STT-RAM as on-chip caches to improve the overall system performance and energy efficiency. A lot of work has been done to address the slow write speed and high write energy on STT-RAM. An energy efficient write termination scheme is proposed in [2]. A dual-write-speed cache design is proposed in [3] to improve the performance while reducing the average STT-RAM write energy consumption. To take advantage of both SRAM and STT-RAM, hybrid cache architectures have been studied and evaluated in recent work [4], [14], [15], [16], [17]. Software assisted techniques have been proposed to enhance the efficiency of hybrid cache [18], [19]. These studies show that caches built with multiple memory technologies have the potential to outperform its counterpart with single technology.

Recently, Smullen et al. [5] propose to relax the non-volatility property of STT-RAM by shrinking the STT-RAM cell surface area. This relaxation can improve the write performance and write energy of STT-RAM. Multi-retention level STT-RAM cache designs are proposed [6] to exploit the optimization space. An application-driven study to determine the retention time is conducted in [7]. They also propose to refresh only dirty data blocks. In [20], a cache-coherence enabled adaptive refresh is proposed to minimize the number of refresh operations for volatile STT-RAM caches in chip multiprocessor systems. In comparison, this paper presents a compilation based approach for refresh minimization for the first time.

8.2 Prior Work on Refresh Schemes

The literature on refresh schemes mainly targeted at DRAM. Previous works can be roughly classified into three categories. The first category exploited the fact that opening a DRAM row causes it to be refreshed, and proposed to associate each DRAM row with a timeout counter to separately track the lifetime of each row [21], [22]. This asynchronous refresh scheme avoids unnecessary refresh operations. All three refresh schemes discussed in this paper exploit similar observations. The second category proposed to mark unused DRAM rows through software and to present them from being refreshed [23]. All three refresh schemes discussed in this paper exploit similar observations by only refreshing valid cache blocks. The third category proposed to take process variation into account, and to refresh DRAM on a finer block-based granularity with refresh intervals varying between blocks [23], [24]. In [25], [26], redundancy is employed to tolerate lower refresh rates, which is orthogonal to this work.

9 Conclusion

This paper proposes a compilation based approach to minimize the number of refreshes in volatile STT-RAM cache by re-arranging the data layout. A heuristic method as well as an ILP formulation is proposed. A large set of experiments are conducted to evaluate the proposed approach under different refresh schemes. These experiments show that the proposed approaches can improve the energy efficiency of volatile STT-RAM cache while maintaining the performance for all refresh schemes.

Acknowledgments

This work was partially supported by a grant from the City University of Hong Kong (Project. 7004048), the National Natural Science Found of China [Project No. 61170022, 61373039, 91118003], and the Specialized Research Fund for the Doctoral Program of Higher Education [Project No. 2013014111002512]. A preliminary version of this work was published in ASP-DAC 2013.

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. Des. Autom. Conf.*, 2008, pp. 554–559.

[2] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for STT-RAM using early write termination," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2009, pp. 264–268.

[3] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, "Design of last-level on-chip cache using spin-torque transfer RAM (STT RAM)," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 3, pp. 483–493, Mar. 2011.

[4] 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. High Perform. Comput. Archit.*, Feb. 2009, pp. 239–249.

[5] C. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 50–61.

[6] Z. Sun, X. Bi, H. H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu, "Multi retention level STT-RAM cache designs with a dynamic refresh scheme," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 329–338.

[7] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs," in *Proc. 49th Annu. Des. Autom. Conf.*, 2012, pp. 243–252.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: Freeman, 1990.

[9] 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. Des. Implementation*, 2005, pp. 190–200.

[10] TI. (2011). Tms320dm368 digital media system-on-chip (DMSoC) [Online]. Available: http://www.ti.com/lit/ds/symlink/tms320dm368.pdf

[11] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power m*core architecture," in *Proc. Workshop Power Driven Microarchit.*, 1998, pp. 145–150.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gen. Optim.*, 2004, pp. 75–86.

[13] LINGO. (2013). Lingo-optimization modeling software for linear, nonlinear, and integer programming [Online]. Available: http://www.lindo.com

[14] 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.

[15] 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. Des.*, 2011, pp. 79–84.

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

[17] Y.-T. Chen, J. Cong, H. Huang, C. Liu, R. Prabhakar, and G. Reinman, "Static and dynamic co-optimizations for blocks mapping in hybrid caches," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Des.*, 2012, pp. 237–242.
[18] Q. Li, J. Li, L. Shi, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted STT-RAM-based hybrid cache for energy efficient embedded systems,"*IEEE Trans. Very Large Scale Integr. Syst.*, vol. 22, no. 8, pp. 1829–1840, Aug. 2014.
[19] K. Qiu, M. Zhao, Q. Li, C. Fu, and C. J. Xue, "Migration-aware loop retiming for STT-RAM-based hybrid cache in embedded systems,"*IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 33, no. 3, pp. 329–342, Mar. 2014.
[20] J. Li, L. Shi, Q. Li, C. J. Xue, Y. Chen, Y. Xu, and W. Wang, "Low-energy volatile STT-RAM cache design using cache-coherence-enabled adaptive refresh,"*ACM Trans. Des. Autom. Electron. Syst.*, vol. 19, no. 5, pp. 5:1–5:13, 2013.
[21] S. P. Song, "Method and system for selective dram refresh to reduce power consumption," Patent US6 094 705 A, Jul. 25, 2000.
[22] M. Ghosh, and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked drams," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2007, pp. 134–145.
[23] T. Ohsawa, K. Kai, and K. Murakami, "Optimizing the DRAM refresh count for merged DRAM/logic LSIs," in *Proc. Int. Symp. Low Power Electron. Des.*, 1998, pp. 82–87.
[24] J. Kim and M. Papaefthymiou, "Block-based multiperiod dynamic memory design for low data-retention power," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 6, pp. 1006–1018, Dec. 2003.
[25] P. G. Emma, W. R. Reohr, and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in dram for cache applications," *IEEE Micro*, vol. 28, no. 6, pp. 47–56, Nov./Dec. 2008.
[26] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-L. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 83–93.

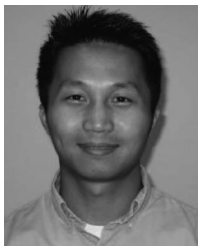
Qingan Li received the BS and PhD degrees from the Computer School of Wuhan University in 2008 and 2013, respectively. He is currently a full-time teacher in the State Key Laboratory of Software Engineering at Wuhan University. His current research interests include compiler optimization, program analysis, and embedded systems.

Yanxiang He received the BS and MS degrees in the Department of Mathematics from Wuhan University in 1973 and 1975, respectively. He received the PhD degree from the Computer School of Wuhan University, in 1999. His research interests include trustworthy software engineering, mobile computing, and distribution computing.

Jianhua Li received the BS degree in computer science from Anqing Teachers' College in 2007 and the PhD degree in computer software and theory from the University of Science and Technology of China in 2013. He is currently a lecturer in the School of Computer and Information at Hefei Institute of Technology. His research interests include on-chip networks, multicore memory system, and emerging non-volatile memories.

Liang Shi received the BS degree in computer science from the Xi'an University of Post and Telecommunication, Xi'an, in July 2008 and the PhD degree from the University of Science and Technology of China, in June 2013. He is currently a full-time teacher in the School of Computer Science at Chongqing University. His research interests include flash memory, embedded systems, and emerging non-volatile memory technology.

Yiran Chen received the BS and MS degrees (both with honor) from Tsinghua University and the PhD degree from Purdue University. After a few years in EDA and data storage industries, he joined the University of Pittsburgh in 2010, where he is currently an assistant professor in ECE Department. His research interests include low-power design, emerging technologies, and embedded system. He is a member of the IEEE.



Chun Jason Xue received the BS degree in computer science and engineering from the University of Texas at Arlington, in May 1997, and the MS and PhD degrees in computer science from the University of Texas at Dallas, in Dec. 2002 and May 2007, respectively. He is currently an assistant professor in the Department of Computer Science at the City University of Hong Kong. His research interests include embedded systems, non-volatile memory, real time systems, and hardware/software codesign.