# Migration-Aware Loop Retiming for STT-RAM-Based Hybrid Cache in Embedded Systems

Keni Qiu, Mengying Zhao, Qingan Li, Chenchen Fu, and Chun Jason Xue

***Abstract*—Recently hybrid cache architecture consisting of both spin-transfer torque RAM (STT-RAM) and SRAM has been proposed for energy efficiency. In hybrid caches, migration-based techniques have been proposed. A migration technique dynamically moves write-intensive and read-intensive data between STT-RAM and SRAM to explore the advantages of hybrid cache. Meanwhile, migrations also introduce extra reads and writes during data movements. For stencil loops with read and write data dependencies, we observe that migration overhead is significant, and migrations closely correlate to the interleaved read and write memory access pattern in a memory block. This paper proposes a loop retiming framework during compilation to reduce the migration overhead by changing the interleaved memory access pattern. With the proposed loop retiming technique, the interleaved memory accesses can be significantly reduced so that migration overhead is mitigated, and energy efficiency of hybrid cache is significantly improved. The experimental results have shown that, with the proposed methods, on average, the migration number is reduced up to 27.1% and the cache dynamic energy is reduced up to 14.0%.**

***Index Terms*—Embedded systems, energy, hybrid cache, loop retiming, migration, STT-RAM.**

## I. Introduction

As TECHNOLOGY scaling continues, conventional SRAM-based caches are facing severe challenges, such as energy consumption and scalability. Recent development of nonvolatile memories (NVMs) has shown that they are qualified candidates for building caches [1]–[8]. Spin-transfer torque RAM (STT-RAM) is one of these candidates. STT-RAM has several advantages, such as high storage density, low leakage power consumption, nonvolatility, fast read speed, and immunity to radiation-induced soft errors [1], [5], [7], [8]. These properties make it an ideal candidate to build caches in embedded systems. However, write operations on STT-RAM have considerably longer latency and higher energy consumption than SRAM. To take advantages of both STT-RAM and SRAM, hybrid cache architecture has been proposed recently [2], [9], [10]. Two major benefits can be obtained from these hybrid cache designs. First, by applying the hybrid cache with higher density than SRAM, the effective cache size can increase significantly under the same chip area constraint. Second, by applying the hybrid cache with much lower static power, the cache dynamic energy consumption can be significantly reduced. To fully explore the advantages of hybrid cache, migration-based techniques are commonly employed to dynamically move write-intensive or read-intensive cache blocks between STT-RAM region and SRAM region [2], [9], [10]. The migration scheme effectively mitigates the drawbacks of long write latency and high write energy of STT-RAM. However, migrations introduce additional reads and writes for data movements. Dynamic energy consumption resulting from this migration overhead is significant in the hybrid cache. This paper addresses the problem of migration overhead through a compiler-assisted optimization approach.

This paper focuses on loops with intensive data array operations, which have read and write data dependencies (R/W dependencies). This class of loops is called stencil loops and they are used to implement relaxation methods in numerical simulations and signal processing. In this paper, two interesting properties are observed for stencil loops on hybrid cache. First, both the migration rate and migration energy overhead are significant. Second, most migrations are closely related to the interleaved read and write access pattern (referred as interleaved access pattern) inside a memory block. Furthermore, the interleaved access pattern corresponds to the R/W dependencies inside a memory block.

Motivated by these observations, this paper proposes a migration-aware loop retiming method to change the interleaved access pattern to reduce the transition events in loops and mitigate the migration overhead. A transition event is caused by a read operation followed by a write, or a write operation followed by a read, in a memory block. Loop retiming is previously applied in a data flow model of a uniform loop to improve the pipeline design for embedded systems with multiple function units [11]–[13]. This paper adopts the loop retiming technique for the novel purpose of transforming the access pattern in a memory block. With the proposed retiming methods, the interleaved access pattern is transformed and migrations can be reduced without any hardware modifications. The contributions of this paper include the following.

1) Two key observations and analysis are presented to analyze the migration properties in stencil loops.
2) Four principles are proposed to quantitatively analyze the relationship of transition events and the interleaved

Manuscript received May 24, 2013; revised August 26, 2013; accepted October 7, 2013. Date of current version February 14, 2014. This work was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China under Grant CityU 123811 and Grant 123210. This paper was recommended by Associate Editor P. R. Panda.

The authors are with the Department of Computer Science, City University of Hong Kong, Kowloon 999077, Hong Kong (e-mail: qiukeni2015@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2013.2288692

access pattern. They reveal loop retiming's impact on transition events considering a single data array.

3) An optimal method and two heuristic methods are proposed to implement migration-aware retiming for the entire loop. Experimental results have shown that the proposed approach can effectively reduce migration overhead and improve energy efficiency of a hybrid cache.

The rest of the paper is organized as follows. The related work is introduced in Section II. In Section III, the observations and analysis of migrations in stencil loops are illustrated. Then, a motivational example using loop retiming to reduce migrations is presented. In Section IV, four principles are derived to reveal loop retiming's impact on the transition events of a single data array. An optimal method and two heuristic methods to conduct retiming for the entire loop are presented in Section V. A set of experiments is conducted to evaluate the proposed methods in Section VI. Finally, Section VII concludes the paper.

## II. Related Work

The related work can be categorized into two areas: STT-RAM-based hybrid cache and loop retiming.

### A. Previous Work on STT-RAM-Based Hybrid Cache

Quite a few techniques have been proposed to address the write speed and energy limitations of STT-RAM [14], [15]. SRAM/STT-RAM hybrid cache hierarchy with 3-D stacking structure has been proposed to improve the cache performance by taking advantages of both the STT-RAM and SRAM [2], [9], [10], [16].

In the hybrid cache, migration-based techniques are proposed to mitigate the drawbacks of STT-RAM and improve the performance and energy efficiency. Sun *et al.* [9] proposed a hybrid cache where the ways in each cache set are composed of a majority of STT-RAM cache lines and a minority of SRAM ones. The main purpose is to keep as many write intensive data in the SRAM region as possible. Data in STT-RAM caches will be migrated to SRAM caches when the successive write operation number reaches a prespecified value. Wu *et al.* [2], [17] evaluated two types of hybrid cache architectures (HCAs), including intercache level HCA and intracache level HCA. In [10], the migration policy includes two kinds of operations. When cache lines are frequently read in SRAM region, they should be migrated to STT-RAM region. Otherwise, when cache lines are frequently written in STT-RAM region, they are migrated to SRAM region.

Since migration would also introduce overhead, migration optimization methods are explored to further improve the efficiency of STT-RAM-based hybrid cache. Li *et al.* [16], [18] proposed the migration-aware compilation (MAC) method to reduce migrations by rearranging data layout. They presented a key observation that migrations are frequently triggered and correlate closely with transition events in memory blocks, which motivates them to place data with consecutively same access operations into the same memory blocks to reduce transition events. The reduction of transition events leads to the reduction of migrations. The MAC method targets scalar data. For data arrays in stencil loops with R/W dependencies, the data elements have both read and write operations. This paper targets migration reduction for these loops.

### B. Previous Work on Loop Retiming

Retiming technique is initially applied to optimize clocked circuits by redistributing delays among the edges so that hardware usage is optimized [19]. Later, it is employed to improve static schedules by rearranging the delays in a uniform loop body for a system with multiple function units. Chao *et al.* [11] proposed combining retiming and unfolding to obtain a static schedule with a reduced average computation time per iteration for 1-D loops. The retiming technique reorganize an iteration by rearranging the dependency delays. The unfolding technique schedules several iterations together. For the case of multidimension loops, Passos and Sha [12] proposed loop retiming in uniform nested loops for full parallelism. Later, Xue *et al.* [13] proposed a method using iterational loop retiming with partitioning to achieve complete memory latency hiding.

All of the previous studies make use of retiming technique for improving software pipelining. In this paper, the retiming technique is employed for the novel purpose of transforming the interleaved access pattern of memory blocks to reduce migrations.

## III. Observations and Motivation

In this section, we analyze the properties of migrations. For the following measurements, we consider a common embedded system configuration with one-level on-chip data cache. Particularly, this data cache is 4-way associative and hybrid, consisting of one way SRAM and three-way STT-RAM. In this hybrid cache, the SRAM region is preferable for write operations to achieve low write dynamic energy and fast write speed, while the STT-RAM region is preferable for read operations to mitigate the pressure of SRAM. We modify the migration policy in [10]. Data in the STT-RAM region is migrated to the SRAM region if they are frequently written, which is referred as w-migration. In the meantime, data in the SRAM region is migrated to the STT-RAM region if they are frequently read, which is referred as r-migration. When there are two consecutive writes or three accumulate writes in the STT-RAM region, a w-migration is triggered. When there are two accumulate reads in the SRAM region, a r-migration is triggered. All of the selected benchmarks have stencil loops. The experiment section has more details of the migration policy and the setup. The migrations and the dynamic energy in these applications exhibit the following properties.

### A. Observation 1: Migration Rate and Migration Energy Overhead

Fig. 1 shows the migration rate, which is defined as the ratio of the number of migrations to the total number of memory accesses. It is found that, on average, 11.2 migrations are triggered per 100 memory accesses.

When compared to [16], it can be seen that the average migration rate in these selected benchmarks is larger than that in the benchmarks in [16]. The reason is that the stencil loops incline to have much more migrations because of the iterative stencil R/W dependencies.

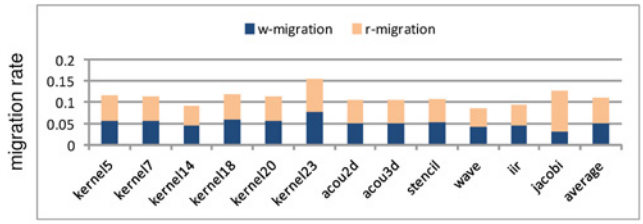



Fig. 1. Migration rate.

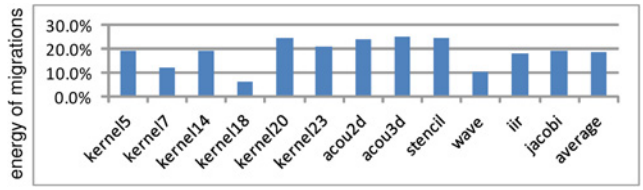



Fig. 2. Migration energy percentage.

We calculate the dynamic energy overhead due to migrations. It is found that, on average 18.7% dynamic energy of the hybrid cache is consumed by the migration operations, as shown in Fig. 2.

### *B. Observation 2: Relationships of Migrations, Interleaved Access Pattern and Data Dependency*

In the previous work, it has been observed that transition-intensive memory blocks are often migration-intensive ones. The number of migrations changes with the number of transition events in memory blocks [16]. Since interleaved accesses of a memory block induce transition events, in the second set of measurements, we further explore the relationship of migrations and interleaved accesses in memory blocks. Fig. 3 shows the correlation coefficient between migrations and interleaved accesses of the benchmarks. This correlation coefficient is computed by migrations occurring at interleaved accesses divided by the total migrations. It is found that the correlation coefficient is of 0.86. In other words, 86% migrations occur in the interleaved accesses of data arrays.

Since the interleaved accesses of a memory block are corresponding to the read and write pairs, they are correlated closely to R/W dependencies, such as read after write (RAW) and write after read (WAR). These data dependencies inside a memory block introduce interleaved reads and writes, resulting in transition events.

On the basis of above observations, the goal of minimizing migrations in STT-RAM-based hybrid cache can be addressed as the problem of mitigating the interleaved accesses inside a memory block. Furthermore, the analysis indicates that the interleaved access pattern correlates closely to the R/W dependencies inside one memory block. These properties, which are visible at compilation time, motivate us to reduce migrations by transforming the R/W dependencies in stencil loops.

### *C. Motivation*

We present a loop reorganizing method to effectively transform the interleaved access pattern by the means of loop

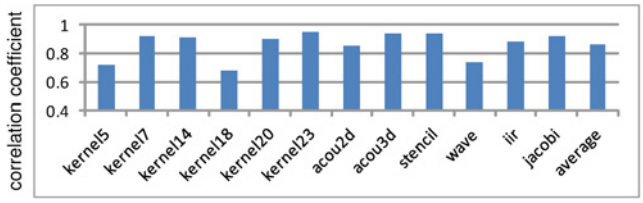



Fig. 3. Correlation coefficient between migrations and interleaved accesses.

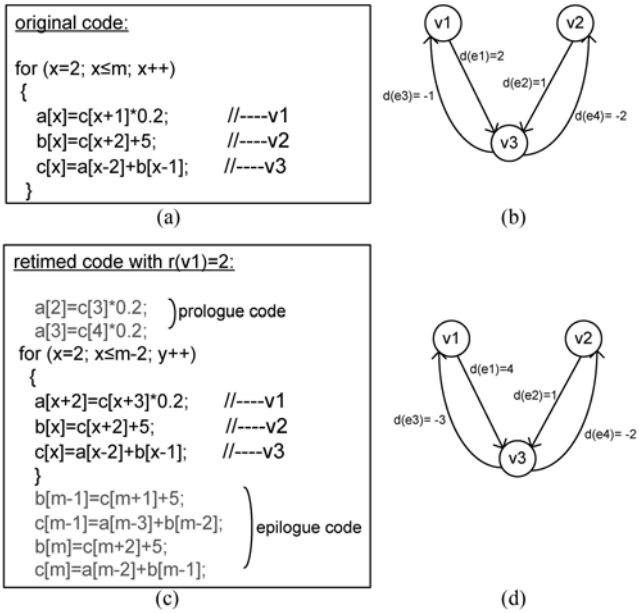



Fig. 4. (a) Original loop code. (b) Original DFG. (c) Retimed code with r(v1) = 2. (d) Retimed DFG.

retiming. First, the loop retiming technique is briefly introduced. Then, a motivational example is presented to show how loop retiming technique mitigates the interleaved accesses inside one memory block and reduce transition events and migrations.

1) *Loop Retiming Preliminary:* Stencil loops present the characteristic of constant dependency vectors, i.e., data dependencies are at a constant distance in the iteration space. An iteration is the execution of the loop body exactly once. The loop body can be represented by a data flow graph (DFG). *Definition 1* (DFG) [12]: A DFG G = <V, E, d> is an edge-weighted directed graph, where V is the set of computation nodes and E is the set of dependency edges. For an edge $u \xrightarrow{e} v$, its delay vector d(e) represents any delay vector between the two computation nodes u and v.

In 1-D DFG (1DFG), the edge delay is defined as an integer value $d(e)=\omega$ rather than a multidimensional vector. In the example of a 1DFG as shown in Fig. 4(a), $d(e1)=2$ implies the value $c[x]$ must wait two iterations before it can use the value $a[x]$.

Retiming allows rearranging the computations of an iteration by moving statements across the original iteration boundary. A retiming vector $r$ is a function that redistributes the nodes in one iteration. The retiming vector $r(u)$ of a node $u$ represents the offset between the original iteration containing $u$ and the one after retiming. Similarly, a retiming vector

in 1DFG is an integer value rather than a multidimensional vector. A legal retiming vector should preserve the original data dependency direction. After a legal retiming, a new DFG $G^r$ is created, and each iteration still has one execution of each node in $G^r$. For the loop in Fig. 4(a), the retimed loop code and the retimed DFG $G^r$ by the retiming vector $r(v1)=2$ are depicted in Fig. 4(c) and (d), respectively. Note that prologue code and epilogue code are needed to set up the initial assignments and complete all the computations in retiming. For the retimed node, all the edge delays connected to it are changed. Its outgoing edge delays add the retiming vector and its incoming edge delays minus the retiming vector.

With retiming, we can reconstruct a new iteration body with different dependency distances, which implies retiming can change the original interleaved read and write accesses.

2) *Migration-Aware Loop Retiming Example:* Considering the loop in Fig. 4(a), there are many interleaved reads and writes in the memory blocks *A, B* and *C* of data arrays *a[x]*, *b[x]* and *c[x]*, respectively. For simplicity, assuming each memory block contains four data elements, we can get the retimed code as depicted in Fig. 4(c) with retiming vector $r(v1)=2$. The detailed computation sequence and data array *A*'s access statistics for the original loop and the retimed loop are illustrated in Fig. 5. Note that only data array *A*'s dependencies and access sequence are shown.

It can be seen from the statistics in Fig. 5(b)–(d) that the interleaved access pattern in a memory block is transformed to a phased access pattern for data array *A*. The original transition event number in one memory block of data array A of three is reduced to one. As a result, the migrations due to large amount of interleaved accesses can be significantly reduced. We are motivated by the example to employ the loop retiming technique to reduce migrations for hybrid caches.

## IV. Migration-Aware Loop Retiming Principles

In this section, we first introduce several definitions and assumptions related to the loop retiming in this paper. Then, we present how loop retiming changes the R/W dependencies distance inside a memory block with respect to a single data array and, thus, changes the interleaved access pattern and transition events.

### A. Foundations

The index bound of a data array is usually much larger than the memory block volume ($v^*$), which represents the number of data elements that can be held in a memory block (or a cache line). Therefore, for multidimension data arrays, only the R/W dependencies of the innermost loop may cause transition events. Thus, only the innermost loop of a DFG is considered in this paper. This paper focuses on 1-D data arrays. The multidimension case with the innermost edge delays of $(0, 0, ...\omega)(\omega \neq 0)$ can be handled accordingly.

In the following, we present notations, terms and assumptions used in this paper.

*Notations:* If $e$ is an edge in a DFG that goes from node $u$ to node $v$, we use the notation $u \xrightarrow{e} v$. In the event that the identity of either the head or the tail of an edge is unimportant, we use the symbol ?, such as $u \xrightarrow{e} ?$ or $? \xrightarrow{e} u$.

*Terms:* In a DFG, if data array *A* is the left-hand item of node *u*, then node *u* is called the *associate node* of data array *A* and all the edges with $u \xrightarrow{e} ?$ are called the *associate edges* of data array *A* or node *u*. Similarly, data array *A* is called the *concerned data array* of node *u* or edge *e*. Any node connected with node *u* is referred as its *neighboring node*.

The term *node number (NN)* denotes the node sequence in the execution of one iteration. For example, in Fig. 4, the node number of node *v2* and node *v3* are $NN(v2)=2$ and $NN(v3)=3$, respectively.

*Assumption 1:* For most embedded systems, there are multiple computation units in a processor to support parallelism. Loop retiming has been applied to reorganize the loop body so that there is no zero delay for any edge in the retimed DFG [12], thus realizing full parallelism. In this paper, the full parallelism is a preprocessing step that is carried out to guarantee that there is no zero delay edge.

*Assumption 2:* It is assumed that the register number is just enough to cover the demand of one iteration. It implies, if a reused data element is not in the same iteration as its initial reference, it is referenced from cache. According to Assumption 1, there is no zero delay edge in the DFG, thus, the read and write operations are deemed as memory load and store operations.

*Assumption 3:* There is at most one write operation with respect to a data array in one iteration of a loop.

We also give two terms of dependency distance and legal delay domain for further discussion.

*Definition 2 [Dependency Distance (DD)]:* The DD represents the distance and in which direction a write operation of a data element is delayed from its read operation along an edge in a DFG.

Assuming data array A is the concerned data array of edge $e$: $u \xrightarrow{e} v$ with $d(e) = \omega$, the DD of data array A along edge $e$ is computed as follows.

*Case 1*: $\omega < 0$: If $NN(u)<NN(v)$, DD=$\omega$. Otherwise, if $NN(u)>NN(v)$, DD=$\omega$-1. $DD<0$ implies a data element's read operation is always executed before its write operation in another iteration along an edge. It is a flow dependency or WAR and referred as WAR dependency.

*Case 2*: $\omega > 0$: If $NN(u)<NN(v)$, DD=$\omega$+1. Otherwise, if $NN(u)>NN(v)$, DD=$\omega$. $DD>0$ implies a data element's write operation is always executed before its read operation in another iteration along an edge. It is an antidependency or RAW and referred as RAW dependency.

Note that, $DD=0$ implies the associate edge has a delay with $d(e)=0$. According to the Assumption 1, this case will not occur.

Since a transition event is defined only in the context of R/W dependencies, the input dependency (RAR) and output dependency (WAW) are both precluded in the DD definition.

In Fig. 4, the original delay of the edge is $DD(e1)=2$ and the retimed delay is $DD^r(e1)=4$ with the retiming vector $r(v1)=2$. Note that the superscript $r$ denotes the retimed value of an item relative to the value before this retiming. A legal retiming on a computation node means that the related data arrays' DD values are changed. However, the DD directions are preserved. This property can be illustrated by the concept of legal delay domain (LDD).

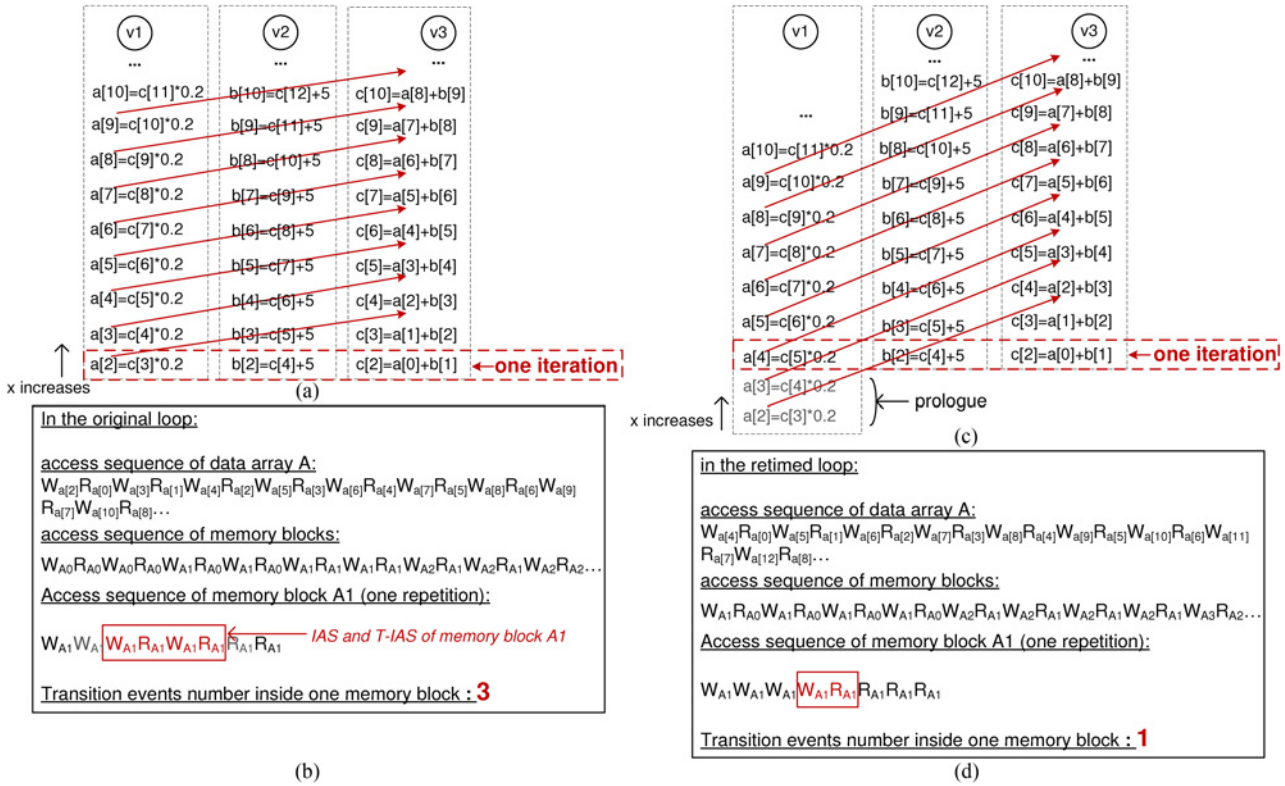



Fig. 5. (a) Original computation sequence of Fig. 4(a). (b) Original access sequence. (c) Retimed computation sequence of Fig. 4(c) with $r(v1) = 2$. (d) Retimed access sequence.

*Definition 3 (LDD):* The LDD represents the set of all possible retimed delays of an edge, each of which can preserve the original dependency.

In other words, if a retimed delay of an edge can preserve the original dependency direction of the concerned data array along the edge, this retimed delay value must belong to the LDD of the edge. If an original edge delay of a positive value delivers RAW dependency, its LDD is *(0,B]* where $B$ denotes the upper bound of the concerned data array. If an original edge delay of a negative value delivers WAR dependency, its LDD is *[-B,0)*. For example, in Fig. 4, the LDDs of the edges are $LDD_{(e1)}$=*(0,m]*, $LDD_{(e2)}$=*(0,m]*, $LDD_{(e3)}$=*[-m,0)* and $LDD_{(e4)}$=*[-m,0)*. The LDD of an edge is determined based on the original DFG and will not change with retiming. A retiming vector for a node is called a legal retiming vector if all the retimed delays of the edges connected the node still belong to their LDDs.

The DD parameter is defined relative to an edge in the DFG. There may be multiple associate edges for a concerned data array. Next, we discuss critical edges of a data array.

*Definition 4 (Critical Edges):* In the current DFG, among all the edges from node u with $u \xrightarrow{e} ?$, the edge with the minimum $|DD|$ in RAW direction is called rAW critical edge $e^{c+}$ of node u (or the concerned data array). Likewise, the edge with the minimum $|DD|$ in WAR direction is called WAR critical edge $e^{c-}$.

Substantially, the two critical edges of a data array indicate the largest transition events in the two dependency directions. For example, in Fig. 6(a), the WAR critical edge and RAW critical edge of data array $A$ (or node $v1$) are $e^{c+} = e1$ and $e^{c-} = e5$, indicated by orange and blue in Fig. 6(b), respectively. If all the associate edges of a data array are in the same dependency direction, the critical edge in the other direction is marked as empty. For example, the WAR critical edge of data array $A$ in Fig. 4(b) is $e^{c+} = \emptyset$ and $e^{c-} = e3$. Note that, the critical edges of a data array are determined based on the current DFG. They may change with the changing of edge delays by retiming.

*Definition 5 (RAW-IAS and WAR-IAS):* For a data array, the RAW-interleaved access segment (RAW-IAS) and the WAR-interleaved access segment (WAR-IAS) represent the subsets of the access reference traces of one memory block along the RAW critical edge and the WAR critical edge, respectively.

RAW-IAS and WAR-IAS provide compact windows to illustrate transition events in the RAW and WAR dependency directions, respectively. For example, Fig. 6(c) depicts the RAW-IAS and WAR-IAS of data array $A$.

### B. Principles

In this subsection, we first derive two theorems to present the quantitative relationship of the transition event number and the associate edge delays with respect to a single data array. Then, we deduce two corollaries to show how loop retiming changes the interleaved access pattern and the transition events.

As discussed in Definition 2, the dependency distance DD along an edge is determined by the edge delay and the node number order of the edge tail and the edge head. We consider two cases of edges as follows. For a data array: we assume

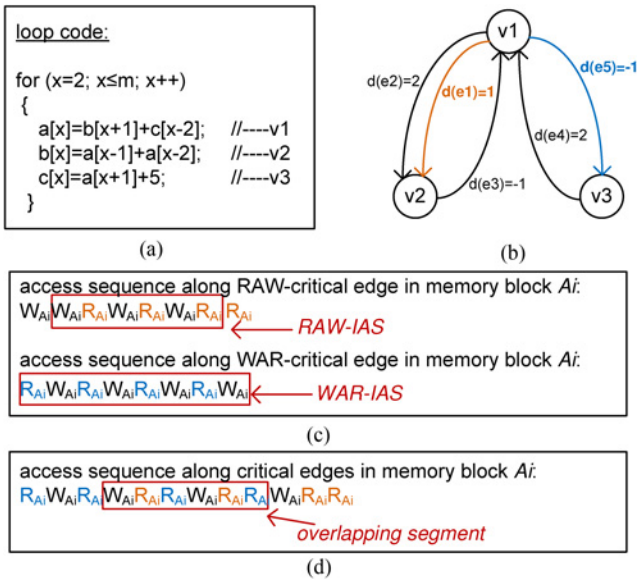



Fig. 6. (a) Loop code. (b) DFG. (c) RAW-IAS and WAR-IAS. (d) Overlapping segment.

1) for each RAW direction edge $e$: $u \xrightarrow{e} v1$, $d(e)=\omega > 0$ and $NN(u)>NN(v1)$ and 2) for each WAR direction edge $e$: $u \xrightarrow{e} v2$, $d(e)=\omega < 0$ and $NN(u)<NN(v)$. Therefore, in both cases, we have $DD = \omega$ according to Definition 2.

*Theorem 1:* In a DFG G=<V, E, d>, it is assumed that node u is the associate node of data array A. For any edge e: if any edge delay $d(e) = \omega$ satisfies $0 < |\omega| < v^*$, then interleaved access pattern must occur in the memory blocks of data array A. Otherwise, if all the associate edge delays $d(e) = \omega$ satisfy $|\omega| \geq v^*$, then there is no interleaved access in the memory blocks of data array A.

*Proof:* For the condition $0 < |\omega| < v^*$, there exist two cases.

*Case 1*: $0 < \omega < v^*$.

Assuming $a[x0]$ is the first element in the memory block $Ai$, the first and last write operations of $Ai$ are $W_{a[x0]}$ and $W_{a[x0+v^*-1]}$, respectively. The first read operation of $Ai$ is delayed from the first write by $\omega$. Since $0 < \omega < v^*$, it can be declared that the first read operation of $R_{a[x0]}$ must occur between the two write operations $W_{a[x0]}$ and $W_{a[x0+v^*-1]}$. That is, the interleaved access pattern occurs in $Ai$.

For example, in Fig. 4(a), assuming $v^* = 4$, we have the access sequence for data array $A$ as shown in Fig. 5(b). As $\omega = 2$ $(0 < \omega < v^*)$, the first and last write operations for the memory block $A1$ are $W_{a[4]}$ and $W_{a[7]}$, respectively. There exists a read operation $R_{a[4]}$ between them, presenting an interleaved access pattern in the memory block.

*Case 2*: $-v^* < \omega < 0$.

Similarly, it can be declared that the first write operation of $W_{a[x0]}$ must occur between the two read operations $R_{a[x0]}$ and $R_{a[x0+v^*-1]}$, presenting an interleaved access.

For the condition $|\omega| \geq v^*$, there also exist two cases: *Case 1*: $\omega \geq v^*$.

The first read operation of $Ai$ is delayed from the first write by $\omega$. Since $\omega \geq v^*$, it can be declared that the first read operation of $R_{a[x0]}$ is accessed after the last write operation of $W_{a[x0+v^*-1]}$. Therefore, the access pattern of the memory block is a phased write sequence followed by a phased read sequence, with no interleaved accesses.

*Case 2*: $\omega \leq -v^*$.

Similarly, it can be declared that the access pattern of the memory block is a phased read sequence followed by a phased write sequence, with no interleaved accesses. ■

*Theorem 2:* In a DFG G=<V, E, d>, data array A is the concerned data array of the node u. It is assumed that data array A's RAW critical edge (if any) is $e^{c+}$ with $d(e^{c+}) = \omega^{c+}$ and its WAR critical edge (if any) is $e^{c-}$ with $d(e^{c-}) = \omega^{c-}$. The transition event number in one memory block of data array A is calculated as follows:

$$tr = \alpha \cdot tr^+ + \beta \cdot tr^- - \alpha \cdot \beta \cdot tr^{\cap} \tag{1}$$

where

$$\alpha, \beta \in \{0, 1\}$$

$$tr^+ = \begin{cases} 2(v^* - \omega^+) + 1 & \text{for } 0 < \omega^+ < v^* \\ 1 & \text{otherwise} \end{cases}$$

$$tr^- = \begin{cases} 2(v^* + \omega^-) + 1 & \text{for } -v^* < \omega^- < 0 \\ 1 & \text{otherwise} \end{cases}$$

$$tr^{\cap} = \begin{cases} 2(v^* + \omega^- - \omega^+ + 1) & \\ \quad \text{for } \omega^+ - \omega^- 1 < v^* \text{ and } 0 < \omega^+, -\omega^- < v^* & \\ 0. \quad \text{otherwise} & \end{cases}$$

the two terms $\alpha$ and $\beta$ indicate whether there exist RAW and WAR critical edges for data array A, respectively. The other three terms $tr^+$, $tr^-$ and $tr^{\cap}$ denote the transition event number in RAW direction and WAR direction, and the overlapped transition event number considering both directions, respectively.

*Proof:* As discussed before, the interleaved accesses in RAW and WAR directions in a memory block can be represented by their RAW-IAS and WAR-IAS, respectively. We calculate the number of transition events of a data array by analyzing its RAW-IAS and WAR-IAS. The RAW critical edge delay is $d(e^{c+}) = \omega^+ > 0$. The RAW critical edge delay is $d(e^{c-}) = \omega^- < 0$. The two terms $\alpha, \beta \in \{0, 1\}$ indicate whether there exist RAW and WAR critical edges, respectively. They can be determined based on the original DFG, because the dependency directions will not change with legal retiming.

There exist four cases:

*Case 1*: $\alpha = 1$, $\beta = 0$.

In this case, there only exists RAW dependency for data array A. If $\omega^+ \geq v^*$, there is no interleaved accesses in the memory blocks according to Theorem 1. The transition event number $tr^+ = 1$, representing the transition of phased write operations to phased read operations. If $0 < \omega^+ < v^*$, interleaved accesses must occur. Assuming $a[x0]$ is the first element in a memory block $Ai$, the first and last transition events are $write \rightarrow read$. In the RAW-IAS, the first write operation is $W_{a[x0+(\omega^+-1)]}$ and the last write operation is $W_{a[x0+(v^*-1)]}$. The number of consecutive $write \rightarrow read$ pairs equals to $x0 + (v^* - 1) - (x0 + (\omega^+ - 1)) + 1 = v^* - \omega^+ + 1$. Thus, the transition event number $tr^+ = 2(v^* - \omega^+ + 1) - 1 = 2(v^* - \omega^+) + 1$.

Therefore, for the data array with only RAW dependencies, the transition events can be expressed as

$$tr^+ = \begin{cases} 2(v^* - \omega^+) + 1 & \text{for } 0 < \omega^+ < v^* \\ 1 & \text{otherwise} \end{cases}$$

*Case 2*: $\alpha = 0,\ \beta = 1$.

In this case, there only exists WAR dependency for data array A. Similarly, if $\omega^- \le -v^*$, we have the transition event number $tr^- = 1$ according to Theorem 1. If $-v^* < \omega^- < 0$, interleaved accesses must occur. The first and last transition events are *read* $\rightarrow$ *write*. In the WAR-IAS, the first read operation is $R_{a[x0+(-\omega^- -1)]}$ and the last read operation is $R_{a[x0+(v^*-1)]}$. The consecutive *read* $\rightarrow$ *write* pair number equals to $(x0+(v^*-1)) - (x0+(-\omega^- -1)) + 1 = v^* + \omega^- + 1$. Thus, the corresponding transition event number $tr^- = 2(v^* + \omega^- + 1) - 1 = 2(v^* + \omega^-) + 1$.

Therefore, for the data array with only WAR dependencies, the transition events can be expressed as

$$tr^- = \begin{cases} 2(v^* + \omega^-) + 1 \text{ for } -v^* < \omega^- < 0 \\ 1 \text{ otherwise} \end{cases}$$

*Case 3*: $\alpha = 1,\ \beta = 1$.

In this case, there exist both RAW dependency and WAR dependency for data array A. It it noteworthy that the writes in the RAW-IAS and WAR-IAS are the same operation sequence, while the reads in the RAW-IAS and WAR-IAS are different operations. As a result, consecutive reads between two writes may occur in the overall access sequence in terms of a memory block as shown in Fig. 6(d). Thus, the transition events in one memory block can be calculated as the sum of $tr^+$ and $tr^-$, but precluding the overlapping segment $tr^\cap$ once if any.

If it satisfies $0 < \omega^+, -\omega^- < v^*$, there exist interleaved accesses in both RAW and WAR directions. In the RAW-IAS of one memory block, the first and last write operations are the $\omega^+th$ and $(v^*-1)th$ writes, respectively. In the WAR-IAS of one memory block, the first and last write operations are the $1st$ and the $(v^* + \omega^- + 1)th$ writes, respectively. Since the writes are the same operations in RAW-IAS and WAR-IAS, the overlapping write/read pair number is $(v^* + \omega^- + 1) - \omega^+$ if $\omega^+ - \omega^- - 1 < v^*$. The corresponding overlapping transition event number is $2(v^* + \omega^- - \omega^+ + 1)$. Otherwise, there is no overlapping write/read pairs.

Therefore, for the data array with both RAW and WAR dependencies, the transition events can be expressed as

$$tr = tr^+ + tr^- - tr^\cap$$

where

$$tr^\cap = \begin{cases} 2(v^* + \omega^- - \omega^+ + 1) \\ \quad \text{for } \omega^+ - \omega^- 1 < v^* \text{ and } 0 < \omega^+, -\omega^- < v^* \\ 0 \quad \text{otherwise} \end{cases}$$

In fact, if it satisfies the conditions $\omega^+ - \omega^- - 1 < v^*$ and $0 < \omega^+, -\omega^- < v^*$, the transition events can be expressed as

$$tr = tr^+ + tr^- - tr^\cap = 2v^*.$$

*Case 4*: $\alpha = 0,\ \beta = 0$.

In this case, node $u$ is an isolated node with no edges in the DFG. Thus, the transition event number is zero.

All together, the transition events can be calculated as (1). Note that we use the transition events occurring in a memory block to represent the transition events of a data array. This is because in stencil loops, the total transition event number of a data array is proportional to the transition event number of one data memory block. ■

*Corollary 1:* In a DFG G=<V, E, d>, data array A is the concerned data array of node u. It is assumed that: 1) $r(u) = \delta_t$, $r(v1) = \delta_{h1}$ and $r(v2) = \delta_{h2}$ are legal retiming vectors for the node $u$, $v1$ and $v2$, respectively, which are applied in the original DFG to obtain the $G^r$; 2) in the retimed DFG $G^r$, if any, the RAW and WAR critical edges of data array A are $u \xrightarrow{e^{c+}} v1$ and $u \xrightarrow{e^{c-}} v2$, respectively; and 3) the delays of the edges $e^{c+}$ and $e^{c-}$ are d(uv1) and d(uv2) respectively before this retiming. Then, the transition event number of data array A in the retimed $G^r$ can be calculated as follows:

$$(tr)^r = \alpha \cdot (tr^+)^r + \beta \cdot (tr^-)^r - \alpha \cdot \beta \cdot (tr^\cap)^r \tag{2}$$

where

$$\alpha, \beta \in \{0, 1\}$$

$$(tr^+)^r = \begin{cases} 2(v^* - (d(uv1) + \delta_t - \delta_{h1})) + 1 \\ \quad \text{for } 0 < d(uv1) + \delta_t - \delta_{h1} < v^* \\ 1 \quad \text{otherwise} \end{cases}$$

$$(tr^-)^r = \begin{cases} 2(v^* + (d(uv2) + \delta_t - \delta_{h2})) + 1 \\ \quad \text{for } -v^* < d(uv2) + \delta_t - \delta_{h2} < 0 \\ 1 \quad \text{otherwise} \end{cases}$$

$$(tr^\cap)^r = \begin{cases} 2(v^* + d(uv2) - d(uv1) + \delta_{h1} - \delta_{h2} + 1) \\ \quad \text{for } 0 < d(uv1) - d(uv2) + \delta_{h2} - \delta_{h1} - 1 < v^* \\ \quad \text{and } 0 < d(uv1) + \delta_t - \delta_{h1}, -(d(uv2) + \delta_t - \delta_{h2}) < v^* \\ 0 \quad \text{otherwise} \end{cases}$$

it can be seen that, in the retimed DFG $G^r$, $d(e^{c+}) = d^r(uv1)$, i.e., $\omega^+ = d(uv1) + \delta_t - \delta_{h1}$, and $d(e^{c-}) = d^r(uv2)$, i.e., $\omega^- = d(uv2) + \delta_t - \delta_{h2}$. Therefore, it is ready to prove (2) based on (1) in Theorem 2.

Corollary 1 presents how to calculate the transition events of a data array if all of its associate node and the neighboring nodes have been retimed. It will be applied in the optimal method and the second heuristic method in Section V.

*Corollary 2:* In a DFG G=<V, E, d>, data array A is the concerned data array of node u. It is assumed that: 1) $r(u) = \delta_t$ is legal retiming vectors for the node $u$ while its neighboring nodes are not allowed for retiming and 2) if any, the RAW and WAR critical edges of node u are $u \xrightarrow{e^{c+}} v1$ with delay d(uv1) and $u \xrightarrow{e^{c-}} v2$ with delay d(uv2), respectively, in the original DFG before this retiming. Then, the transition event number in a memory block of data array A in the retimed DFG $G^r$ with $r(u) = \delta_t$ can be calculated as follows:

$$(tr)^r = \alpha \cdot (tr^+)^r + \beta \cdot (tr^-)^r - \alpha \cdot \beta \cdot (tr^\cap)^r \tag{3}$$

where

$$\alpha, \beta \in \{0, 1\}$$

$$(tr^+)^r = \begin{cases} 2(v^* - (d(uv1) + \delta_t)) + 1 \text{ for } 0 < d(uv1) + \delta_t < v^* \\ 1 \quad \text{otherwise} \end{cases}$$

$$(tr^-)^r = \begin{cases} 2(v^* + (d(uv2) + \delta_t) + 1 \text{ for } -v^* < d(uv2) + \delta_t < 0 \\ \quad 1 \text{ otherwise} \end{cases}$$

$$(tr^\cap)^r = \begin{cases} 2(v^* + d(uv2) - d(uv1) + 1) \\ \text{for } 0 < d(uv1) - d(uv2) - 1, d(uv1) + \delta_t, -(d(uv2) \\ \quad + \delta_t) < v^* \\ 0 \quad \text{otherwise} \end{cases}$$

Since node $u$ is retimed with $r(u) = \delta_t$ while its neighboring nodes are not allowed for retiming, it implies *r(?)=0* for any

$u \xrightarrow{e} ?$ or $? \xrightarrow{e} u$. It can be deduced that the RAW and WAR critical edges of data array $A$ are kept unchanged after the retiming with $r(u)$. In the retimed DFG $G^r$, the RAW critical edge delay is $d(e^{c+}) = d^r(uv1)$, i.e., $\omega^+ = d(uv1) + \delta_t$ and the WAR critical edge delay is $d(e^{c-}) = d^r(uv2)$, i.e., $\omega^- = d(uv2) + \delta_t$. Therefore, Corollary 2 can be proved based on Corollary 1 when setting $r(v1) = \delta_{h1} = 0$ and $r(v2) = \delta_{h2} = 0$.

Corollary 2 shows how to calculate the transition events of a data array if only its associate node has been retimed while the neighboring nodes have not been allowed for retiming. It will be applied in both of the heuristic methods in Section V.

For the other cases of edge delay directions and the node number order, such as the case of $d(e)=\omega > 0$ and $NN(u)<NN(v1)$, or the case of $d(e)=\omega < 0$ and $NN(u)>NN(v2)$, the corresponding transition event number calculations can be similarly derived.

## V. Migration-Aware Loop Retiming Algorithm

The above discussions focus on how to implement a legal retiming and how retiming impacts the transition events considering an individual data array. This section proposes two approaches to solve the retiming vectors for the entire loop to effectively reduce the overall transition events. The first is an optimal approach. The problem of determining the optimal retiming vectors for each node to minimize the overall transition events is formulated. The second is the heuristic approach, including two heuristic methods.

### A. Optimal Retiming

The retiming problem for the entire loop can be addressed as determining a legal retiming vector for each node so as to achieve the minimum overall transition event number. As discussed before, retiming may enlarge the dependency distance length of a data array. As a result, data locality will be influenced and in turn data reuse in the cache might be spoiled. Moreover, retiming introduces prologue and epilogue code and code size expansion increases with the absolute values of retiming vectors. Therefore, the retimed dependency distances for each data array should be as small as possible. The problem can be formulated as follows.

Objective 1 (high priority)

$$Minimize \sum_{all\ data\ arrays} \alpha \cdot (tr^+)^r + \beta \cdot (tr^-)^r - \alpha \cdot \beta \cdot (tr^\cap)^r.$$

Objective 2 (low priority)

$$Minimize \sum_{all\ data\ arrays} \alpha \cdot max(DD^{RAW})^r - \beta \cdot max(DD^{WAR})^r$$

subject to: each retimed edge delay: $d^r(e) = \omega + \delta_t - \delta_h \in LDD_{(e)}$.

The values of $\alpha$ and $\beta$ for each data array and the LDD for each edge can be determined based on the original DFG. In the constraints, $\omega$ denotes the original delay of edge $e$ before retiming, i.e., $d(e) = \omega$, and $\delta_t$ and $\delta_h$ denote the retiming vectors for the tail node and the head node of edge $e$, respectively.

Since there are two objectives for the problem, we adopt a two-step solver to find the overall optimal solution. In the first step, Objective 1 with the constraints can be formulated as several sub-integer linear programming (sub-ILP) problems based on different final retimed critical edge cases. The global minimal transition event number can be obtained by considering all sub-ILP solutions.

In the second step, Objective 2 is treated as the final objective. The terms $max(DD^{RAW})^r$ and $max(DD^{WAR})^r$ denote the maximum retimed dependency distance lengths in the RAW direction and the WAR direction, respectively. Objective 2 can also be transformed into several sub-ILP problems based on different final retimed maximal edge delay cases. The original constraints together with the result of the Objective 1 are treated as final constraints. By solving these sub-ILP problems, the final optimal retiming solution with minimal total transition events and minimal total dependency distance lengths can be found. For the case in Fig. 4, assuming $v^*=8$, the optimal retiming solution is: $r(v1)=6$, $r(v2)=7$ and $r(v3)=0$.

### B. Heuristic Retiming

The optimal retiming algorithm is time consuming to run when the node number and/or loop bounds are large. In this section, we present two heuristic algorithms to find satisfactory retiming solutions with very low time complexity. In the heuristic retiming, retiming is conducted for the nodes in the DFG one by one.

First, we introduce the concept of optimal retiming vector for each node, which is applied in the heuristics. It is expected that the retimed associate edge delays for each node deliver minimal transition events, while keeping as small dependency distance lengths as possible. Generally, small retimed dependency distance lengths imply good temporal locality, and small prologue and epilogue code size. With these considerations, the independent optimal retiming vector is proposed to find the best retiming vector for each node.

*Definition 6* [*Independent Optimal Retiming Vector (IORV)*]: The IORV represents the unique retiming vector among all the legal retiming vectors of a node, with which, the transition event number of the concerned data array is minimized and the overall transition event number of the loop is reduced. Meanwhile, the retimed associate edge(s) deliver minimum dependency distance length $|DD|$.

The underlying concept of IORV is to maximize the retiming's positive impact on transition event reduction and minimize the potential negative impact on locality and code size expansion. It is noteworthy that the IORV of a node is calculated on a basis that the retiming vectors of all its neighboring nodes are zeros. The IORV calculation algorithm is shown in Algorithm 1. It does not traverse all the legal retiming vectors of a node when searching the IORV, because the interleaved accesses will disappear when the retimed $|DD|$ goes beyond the bound $v^*$ (line 3). After each legal retiming of a node, we calculate the retimed transition events $tr$ of the concerned data array according to Corollary 2. The final IORV of a node is the retiming vector that delivers the minimum transition events of the concerned data array. Meanwhile, the overall transition events are also reduced by this retiming with the IORV (line 11). For the case in Fig. 4(a), $r^{IORV}(v1) = 2$.

The IORVs for the nodes are independently determined based on the DFG. However, they cannot be directly and simultaneously applied. The reason is that retiming one node may influence the other nodes' retiming decisions. For

**Algorithm 1** IORV Calculation

**Input:**
DFG of a loop;
$u$: the associate node of data array $A$;
$tr$: the current transition event number of data array A;
$tr_{min}$: the minimum transition event number of data array $A$;
$TR$: the overall transition event number of the loop;
$R$: the searching range when calculating the IORV;

$r^{IORV}(u) = 0$; $tr_{min} = -1$; $TR = -1$
**for** each associate edge $e$ of node $u$ **do**
  Calculate $R$: $R = (0, v^*)$ for an RAW edge or $R = (-v^*, 0)$ for an WAR edge;
  **for** each possible retimed $d^r(e)$ in its $R$ **do**
    Calculate a tentative retiming vector of node $u$: $rv = d^r(e) - d(e)$;
    **if** all the other retimed delays of the edges connected to node $u$ belong to their $LDDs$ when tentatively retiming with $r(u) = rv$ **then**
      Calculate the retimed transition event number $(tr)^r$ of data array $A$ and the retimed overall transition event number $(TR)^r$ of the loop with $r(u) = rv$ according to Corollary 2;
      **if** $tr_{min} = -1$ and $TR = -1$ **then**
        $tr_{min} = (tr)^r$; $TR = (TR)^r$;
      **else**
        **if** $tr_{min} > (tr)^r$ and $TR > (TR)^r$ **then**
          $r^{IORV}(u) = rv$; $tr_{min} > (tr)^r$; $TR > (TR)^r$;
        **end if**
      **end if**
    **end if**
  **end for**
**end for**
**return** the final $r^{IORV}(u)$;

**Algorithm 2** PMBR Algorithm

**Input:**
DFG of a loop; $N$: the set of all the active nodes in the DFG;
$Nu$: the set of node $u$'s neighboring nodes;
$num(Nu)$: the element number of the set Nu;
$state(u)$: the $active/inactive$ state of node $u$, indicating whether node $u$ is available for retiming;

$N = \emptyset$;
Put all the nodes in the DFG into the set $N$;
Set each node in $N$ an $active$ state;
**repeat**
  **for** each node in $N$ **do**
    Calculate the IORV according to Algorithm 1;
  **end for**
  $Nu = \emptyset$;
  Pick up the node $u$ whose concerned data array has the largest transition event reduction after the retiming with its IORV;
  Retime node $u$ with $r(u)^{IORV}$ and update the DFG;
  $state(u) = inactive$; $N = N$-{$u$};
  Put the neighboring nodes of node $u$ into the set $Nu$;
  **for** each node $v$ in $Nu$ **do**
    **if** $state(v) = active$ **then**
      $state(v) = inactive$;
      $N = N$-{$v$};
    **end if**
  **end for**
**until** $N = \emptyset$;
**return** the final updated DFG;

example, using Corollary 2, we can calculate the IORVs $r^{IORV}(v1) = 2$, $r^{IORV}(v2) = 3$ and $r^{IORV}(v3) = -3$ in Fig. 4. If we first retime node *v1* with $r^{IORV}(v2) = 2$, then the retimed edge delays are $d^r(e1) = 4$, $d^r(e2) = 1$, $d^r(e3) = -3$, and $d^r(e4) = -2$. It is obvious that the original $r^{IORV}(v3)$ is not applicable any more. Then, we should calculate a new IORV $r^{IORV}(v3) = -2$ based on the retimed DFG. But in turn, this new $r^{IORV}(v3)$ changes the previous retiming effect and further influences the retiming decision of node *v2*.

In order to simplify the problem, we propose two heuristic approaches. One is conducted based on the lockdown rule and the other is conducted in the relaxed edge delay set (REDS) domain for each retimed data array. Both of them adopt the IORV of each individual data array for retiming.

*Heuristic 1 (Retiming Based on Lockdown Rule):* In this method, once a computation node is retimed, its neighboring nodes are locked down to avoid receiving further retiming. We assign an active state attribute for each node initially. When a node is in active state, it is available for retiming. When the retimed node is set to inactive state, all its neighboring nodes will also be set to inactive states and cannot be retimed any more. This lockdown rule protects the former retiming effectiveness and drives the retiming process for the entire loop.

In terms of how to determine the priority for the nodes to get retimed, this paper proposes two algorithms: priority with the most benefit retiming algorithm (PMBR algorithm) and priority with the least impact retiming algorithm (PLIR algorithm). In the PMBR algorithm, we select the active computation node that can achieve the most transition event reduction with its IORV for each retiming determination. The PMBR algorithm is shown in Algorithm 2.

Since the neighboring nodes of the retimed node would also be locked down with the lockdown rule, their retiming opportunities are deprived as well. We expect the locked node number to be, as small as possible on each retiming decision. The PLIR algorithm is proposed based on this consideration, in which each time we select the active computation node with the least neighborhood number. It is very similar to the PMBR algorithm and thus, the detailed explanation is omitted here. The time complexity of these IORV based heuristic algorithms is $O(N^2 \cdot E)$ where $N$ and $E$ denote the computation node number and edge number, respectively, in the DFG.

*Heuristic 2 (Retiming Based on REDS):* The lockdown rule based heuristic algorithms are implemented on the two foundations: 1) IORVs are employed to retime the concerned nodes and 2) the lockdown rule is applied to freeze the neighboring nodes once a node is retimed. Actually, we may relax the second constraint to involve every node in the DFG into retiming consideration and achieve more transition event reduction. Thus, we propose a REDS-based heuristic retiming method.

*Definition 7* [*RAW-REDS and WAR-REDS*]: If the concerned data array of node u has RAW dependencies, its RAW-REDS represents the set of retimed delays of the associate edges in RAW direction, in which the RAW-critical edge has lager $|DD|$ after retiming. Likewise, if the concerned data array of node u has WAR dependencies, its WAR-REDS represents the set of retimed delays of the associate edges in WAR direction, in which the WAR-critical edge has lager $|DD|$ after retiming.

The definition indicates, if any updated critical edge delay of a node after retiming another node belongs to the REDS domain, the retimed critical edge will introduce a larger $|DD|$. It can be deduced that: 1) any retimed delay in REDS definitely belongs to its LDD and 2) each retimed critical dependency distance satisfies $DD^r(e^{c+}) \geq DD(e^{c+})$ after retiming with a vector in RAW-REDS (if any) and $DD^r(e^{c-}) \leq DD(e^{c-})$ after retiming with a vector in WAR-REDS (if any). The REDS of a data array is determined based on the current DFG and varies with the critical edges. For example, in Fig. 4(d), the RAW-REDS of node *v1* after the first retiming with *r(v1)=2*

**Algorithm 3** REDS Based Retiming Algorithm

**Input:**
DFG of a loop; $N$: the set of all the active nodes in the DFG;
$state(u)$: the $active/inactive$ state of each node $u$, indicating whether node $u$ is available for retiming;

```
1: N = ∅;
2: Put all the nodes in the DFG into the set N;
3: Set each node in N an active state;
4: repeat
5:   for each node in N do
6:     Calculate the IORV according to Algorithm 1;
7:   end for
8:   Pick up the node u whose concerned data array has the largest transition
     events reduction after the retiming with its IORV;
9:   for each neighboring node v of node u do
10:    Calculate the RAW-REDS and WAR-REDS of node v;
11:    for each edge e: v --e--> u do
12:      Calculate the transition event number of the concerned data array
         of node u according to Corollary 1 when tentatively conducting
         the retiming with r^IORV(u);
13:    end for
14:    if state(v) = inactive then
15:      //if node v has received retiming before
16:      if its associate retimed edge delays belong to the corresponding
         REDS then
17:        //this retiming for node u can be finalized
18:        Update the DFG with r^IORV(u);
19:        state(u) = inactive; N = N − {u};
20:      end if
21:    else
22:      //If node v has not been retimed before, this retiming for node u
         can be finalized.
23:      Update the DFG with r^IORV(u);
24:      state(u) = inactive; N = N − {u};
25:    end if
26:  end for
27: until N = ∅;
28: return the final updated DFG;
```

is *[4,m]*. In order to maintain the retiming efficiency of node *v1*, the later retiming should guarantee that, the retimed RAW critical edge delay $d^r(e1)$ belongs to the domain *[4,m]* after retiming other nodes. Thus, the transition event reduction from retiming *r(v1)=2* will not be cut down by the latter retiming.

The REDS based retiming method does not adopt the lockdown rule to prevent the interferences imposed on the former retimed ones by the latter retimings. It is conducted on the basic idea that, as long as each of the retimed data arrays' transition event number does not get larger when tentatively conducting a retiming with its IORV for a new node, this new retiming can be finalized. On each retiming, the previous retimed nodes' corresponding transition events shall be calculated again according to Corollary 1. This constraint ensures that each new retiming will not have negative impact on the previous transition event reduction. The detailed retiming process is shown in Algorithm 3. In the algorithm, each time the active node with the most transition events reduction with its IORV has the highest priority for retiming. Once a node is retimed, it will be set into an inactive state and cannot be considered for retiming any more. However, its neighboring nodes are still active for possible further retiming. The time complexity of the REDS based retiming algorithm is $O(N^2 \cdot E)$.

## VI. Experiments

In this section, we first introduce the migration policy applied in the evaluation and the experimental setup including benchmark characteristics and architecture parameters. Then, a discussion based on the experimental results is presented. We also study the impact of threshold selection on the proposed method. At last, the effectiveness of the proposed method is analyzed by comparing our method to the MAC method [16].

### A. Migration policy

We adopt the migration policy in [10]. Their work targets chip multiprocessors (CMPs) and considers neighborhood group caching (NGC), while our work focuses on embedded systems with one-level cache. So the intercore migration strategy in [10] is not included. In the experimental evaluation, the migration policy is as follows.

1) A two-bit w-migration counter and a one-bit cross-access counter are extended for each line in STT-RAM to control the line migration to SRAM. Likewise, a two-bit r-migration counter is extended for each line in SRAM to control the line migration to STT-RAM. Once a line is loaded with a new data block, the corresponding counters will be preset to zero.
2) r-migration: The r-migration prefetches the read-intensive contents from SRAM to STT-RAM to mitigate the pressure of SRAM. If a read request hits a cache line on SRAM, the r-migration counter is increased by 1. When it reaches the r-migration threshold *rc-th* '10', the corresponding cache line will be migrated into the STT-RAM region. In the migration, the least recently used (LRU) line in the STT-RAM region within the same cache set is selected as the potential destination to be replaced. If the replaced line in the STT-RAM region is valid, we choose to swap the two lines in SRAM and STT-RAM. Otherwise, it is written back to the main memory.
3) w-migration: The w-migration migrates lines from STT-RAM to SRAM to save write energy. If a write request hits a cache line on STT-RAM, the w-migration counter is increased by 1. If the cross-access counter is 0 and the w-migration counter reaches the w-migration threshold *seq-wc-th* '10', the line will be migrated into SRAM. If the cross-access counter is 1 and the w-migration counter reaches the w-migration threshold *nonseq-wc-th* '11', the line will also be migrated into SRAM. The cross-access counter of a line is set to 1 when this line has been cross accessed by write and read operations. Since the SRAM region is the minority in the hybrid cache, the role of the cross-access counter is to make sure that the intensive write trend is obvious enough, such that the w-migration is meaningful. In the migration, the LRU line in the SRAM region within the same cache set is selected as destination to be replaced. If the replaced line in the SRAM region is valid, we choose to swap the two lines in STT-RAM and SRAM. Otherwise, it is written back to the main memory.
   We will vary the thresholds and study the threshold sensitivity in Section VI-D.

### B. Experimental Setup

The proposed retiming technique is implemented based on the LLVM compiler [20]. Five versions of executables are generated: the default version with no loop retiming, the retimed

TABLE I
BENCHMARK DESCRIPTIONS

| Test set | Benchmark | Description |
|---|---|---|
| Livermore | kernel 5 | tri-diagonal elimination, below diagonal |
| | kernel 7 | equation of state fragment |
| | kernel 14 | particle in cell |
| | kernel 18 | explicit hydrodynamics fragment |
| | kernel 20 | discrete ordinates transport |
| | kernel 23 | implicit hydrodynamics fragment |
| BLITZ++ | acou2d | acoustic2d |
| | acou3d | acoustic3d |
| | stencil | array stencil |
| DSP programs | wave | wavefront computation |
| | iir | infinite impulse response |
| | jacobi | HPF Jacobi kernel source |

TABLE II
BENCHMARK STATISTICS

| Benchmark | data arrays | instructions | reads | writes |
|---|---|---|---|---|
| kernel 5 | 7 | 1.90E+06 | 9.35E+05 | 2.17E+05 |
| kernel 7 | 10 | 5.80E+06 | 3.03E+06 | 3.17E+05 |
| kernel 14 | 6 | 3.35E+05 | 1.30E+05 | 5.91E+04 |
| kernel 18 | 9 | 1.99E+08 | 8.15E+07 | 4.52E+06 |
| kernel 20 | 5 | 9.30E+06 | 2.13E+06 | 6.17E+05 |
| kernel 23 | 8 | 9.05E+05 | 4.35E+05 | 2.17E+05 |
| acou2d | 6 | 8.97E+07 | 3.48E+07 | 3.10E+07 |
| acou3d | 6 | 5.91E+08 | 2.33E+08 | 2.12E+08 |
| stencil | 6 | 8.18E+08 | 3.22E+08 | 2.93E+08 |
| wave | 3 | 1.03E+05 | 3.40E+04 | 1.68E+04 |
| iir | 3 | 2.51E+05 | 9.04E+04 | 3.99E+04 |
| jacobi | 4 | 5.65E+07 | 2.93E+07 | 7.33E+06 |

version with the optimal algorithm (referred as *optimal* version), the retimed version with the PMBR heuristic algorithm (referred as *heu1-PBMR* version), the retimed version with the PLIR heuristic algorithm (referred as *heu1-PLIR* version) and the retimed version with the REDS based heuristic algorithm (referred as *heu2* version). All of the versions are compiled with 'O3' option.

The benchmarks are selected from Livermore [21], BLITZ++ [22], and DSP programs. All of them include the stencil loops with R/W data dependencies. The benchmark descriptions are shown in Table I. The benchmark statistics are shown in Table II.

A Pin-based [23] simulator of STT-RAM based hybrid cache with the migration policy introduced in Section VI-A is designed for evaluation. The target architecture is depicted in Table III. The cache and memory parameters are obtained from a modified version of CACTI [24].

### C. Results

We evaluate the four retimed versions with respect to the metrics of migration number, write operation number to STT-RAM, dynamic energy of the hybrid cache, memory access latency and code size expansion overhead. Then, we compare the effectiveness of the algorithms. All the results are normalized to those of the default version.

1) *Migrations:* The reduction of migrations is shown in Fig. 7. On average, the migration number is reduced by 27.1% with the optimal algorithm and 21.8% with the heuristic algorithms. It can be seen that all the proposed methods can significantly reduce migrations.

TABLE III
ARCHITECTURE PARAMETERS

| Parameter | Description |
|---|---|
| processor | single core |
| hybrid data cache | 32KB, 32B line size, LRU, 4-way<br>1-way for SRAM (8KB)<br>3-way for STT-RAM (24KB)<br>write-allocate policy: write back<br><br>SRAM access latency: 6 cycles<br>SRAM access dynamic energy: 0.388nJ<br>STT-RAM R/W latency: 6/28 cycles<br>STT-RAM R/W dynamic energy: 0.4/2.3nJ |
| main memory | latency: 300 cycles |

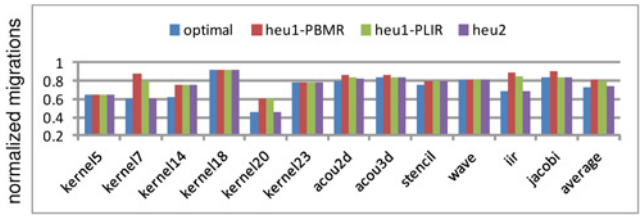



Fig. 7. Normalized migrations.

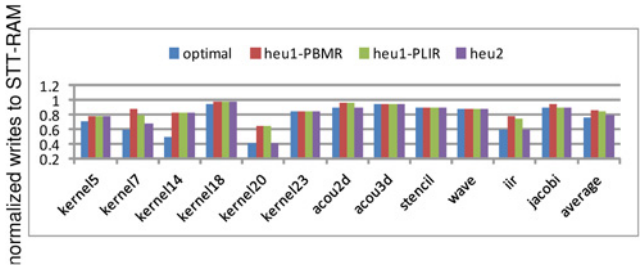



Fig. 8. Normalized writes to STT-RAM.

For the benchmark *kernel 20*, the migration reduction is significant. The reasons lie in two aspects. First, the original migration rate and correlation coefficient are both very high. Second, most of the nodes in the loop have received effective retiming. Therefore, the interleaved accesses and transition events are significantly reduced. The benchmark *kernel 18* does not have such a big migration reduction. The reason is that, its original migration correlation coefficient is not so high, resulting in limit migration reduction from loop retiming. Although the benchmarks *acou3d*, *stencil*, and *jacobi* have very high correlation coefficients, their results have not shown so much improvement. The main reason is that, some data arrays in them have both RAW and WAR edges and their transition events reduction is limited under the legal retiming constraint.

2) *Writes to STT-RAM:* The normalized write operations to STT-RAM region is shown in Fig. 8. On average, the write operations on STT-RAM are reduced by 22.3% with the optimal method.

It is observed that the write to STT-RAM reduction is consistent with the migration reduction. This is because the original interleaved accesses in stencil loops introduce a lot of migrations, resulting in plenty writes to STT-RAM. When

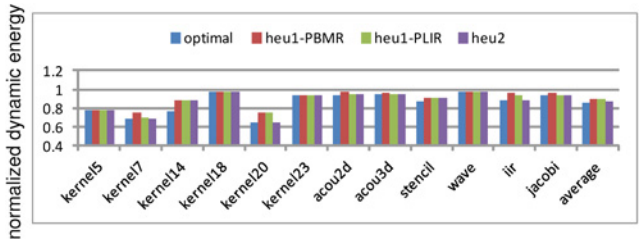



Fig. 9. Normalized dynamic energy.

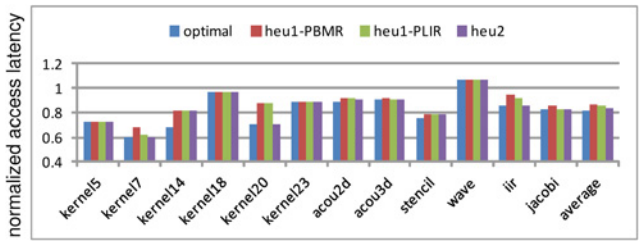



Fig. 10. Normalized memory access latency.

the migrations are mitigated by the proposed loop retiming methods, the energy and time consuming writes to STT-RAM are reduced. This reduction of writes comes from two aspects. First, the writes to STT-RAM caused by migrations are reduced. Second, the writes to STT-RAM before migration action are also reduced as the access pattern transforms.

3) *Dynamic Energy:* To evaluate the energy efficiency, we only consider the dynamic energy consumed by the hybrid cache, precluding the dynamic energy consumed by the main memory. As shown in Fig. 9, the total dynamic energy is reduced up to 14.0%. The improvement mainly results from the reduction of writes to STT-RAM.

For the benchmark wave, its dynamic energy reduction is not consistent with the reduction of writes to STT-RAM. The reason is possibly that, the cache misses are increased because retiming enlarges data dependency distance lengths and some of the original reuses in the cache may be destroyed. In turn, the additional misses introduce more memory block loading from the main memory to the cache. This cost due to additional cache misses partially cuts off the net-benefit of energy reduction from writes reduction to STT-RAM.

4) *Memory Access Latency:* We estimate the memory access latency by calculating the execution cycles of reads and writes occurring in the STT-RAM region, the SRAM region, and the main memory, using the cache and memory parameters depicted in Table III.

It is interesting that the benchmark *kernel 7* has significant improvement in memory access latency though it has not so much migration reduction. The reason is that the cache miss rate is very small and the STT-RAM write latency has a big fraction in the total access latency. With the reduction of writes to STT-RAM by loop retiming, the total access latency is greatly reduced.

5) *Code Size Expansion Overhead:* The proposed retiming techniques may lead to code size increasing. The code size expansion overhead is mainly caused by the additional prologue/epilogue code generated by retiming. It is related to the retiming vectors for the nodes in the DFG. The retimed

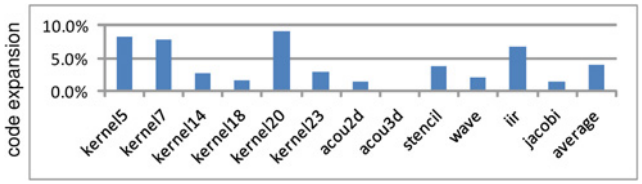



Fig. 11. Code size expansion percentage.

TABLE IV
THRESHOLD CONFIGURATIONS

| Threshold | Case 1 | Case 2 | Case 3 | Case 4 |
|---|---|---|---|---|
| *seq-wc-th* | 1 | 2 | 4 | 6 |
| *nonseq-wc-th* | 2 | 3 | 5 | 8 |
| *rc-th* | 1 | 2 | 4 | 6 |

code size with the *heu2* algorithm is slightly increased by around 4.0% as shown in the Fig. 11. It can be deduced that the proposed loop retiming approach has little impact on the code size. The main reason is that the retiming vectors and the node number are both limited. Moreover, we have taken the code size expansion issue into special consideration when formulating the optimal vectors and determining the IORVs.

6) *Algorithm Comparison:* For most benchmarks, the results of the *heu1-PMBR*, *heu1-PLIR*, and *heu2* algorithms are pretty close as shown in Figs. 7–10. The reason is that the final retiming vectors determined by these algorithms are the same. For the cases that PLIR outperforms PMBR, the cause is that PLIR involves more nodes for retiming. It implies migrations in more data blocks are reduced. For the cases that *heu2* outperforms the two *heu1* algorithms, it can be speculated that more nodes can obtain benefit from their retiming by relaxing the lockdown rule. For several benchmarks, the heuristic methods exhibit the same or close performance compared to the optimal method. It is a good indication to apply the heuristic loop retiming methods in real embedded systems.

### D. Threshold Sensitivity

In the migration policy introduced in Section VI-A, there are three kinds of thresholds: *seq-wc-th*, *nonseq-wc-th*, and *rc-th*, denoting the sequential write threshold value, nonsequential write threshold value and read threshold value, respectively. It means, when the writes or reads reach the threshold values, the corresponding migration will be triggered. We vary the threshold values from Case 1 to Case 4 as shown in Table IV. The cross-access counter is kept one-bit. Note that, Case 2 is the default setup applied in the above experiments.

The migration rate and dynamic energy consumption results for these cases are illustrated in Fig. 12. The dynamic energy results with the *heu2* algorithm are normalized to those under the default setup of Case 2. The migration rate results show that all the benchmarks are greatly sensitive to the threshold values. This can be readily explained that larger threshold values make more conservative choices and introduce less migrations, while smaller threshold values make more aggressive choices and introduce larger migrations.

However, the normalized dynamic energy results show different trends for different benchmarks. The benchmarks

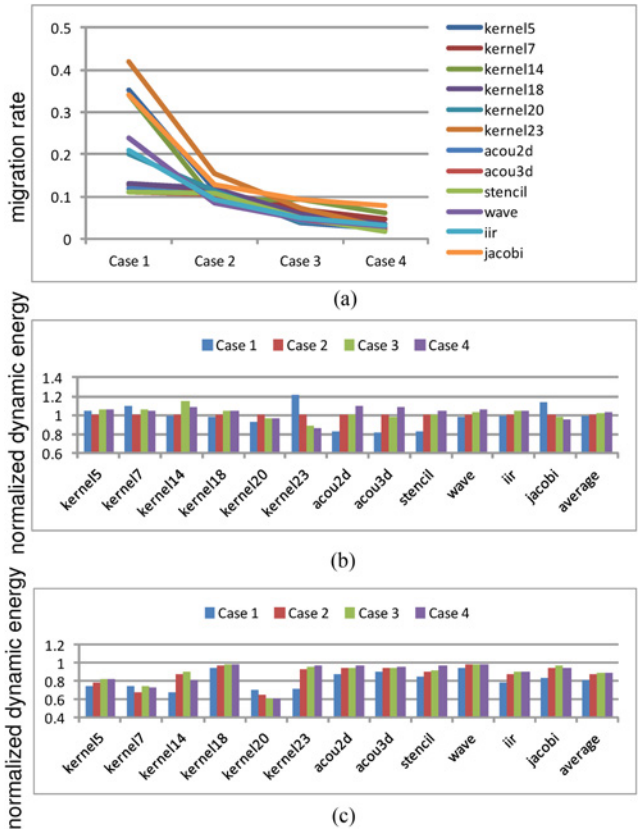



Fig. 12. (a) Migration rate. (b) Dynamic energy normalized to Case 2. (c) Dynamic energy normalized to the default version with no retiming under different threshold setups.

*kernel 20* and *iir*, prefer the threshold values in Case 1, and *kernel5*, *kernel7*, *kernel14* prefer the threshold values in Case 2, while *acou2d*, *acou3d* and *jacobi* prefer larger threshold values. This can be explained as follows. Smaller threshold values imply larger migrations. On one hand, it is promising to obtain more STT-RAM writes reduction from the migration technique. On the other hand, more STT-RAM writes will be introduced due to the frequent migrations themselves. Therefore, the final results present various trends.

Furthermore, we evaluate the effectiveness of the proposed loop retiming technique under these four threshold configurations. The dynamic energy results with the *heu2* algorithm are normalized to those with no loop retiming, as shown in Fig. 12. The energy improvement is determined by two combined factors: the energy reduction and the original energy. We have two observations on the results. First, the loop retiming method can effectively reduce cache dynamic energy for all the benchmarks. Second, there is no obvious trend of preferring larger threshold values or smaller ones. In any case, two trends can be deduced as follows. Smaller threshold values imply more migrations, hence more dynamic energy reduction may be obtained by loop retiming. If the original energy is small, the loop retiming technique will deliver more energy saving under the case of small threshold values. Likewise, larger threshold values imply less dynamic energy reduction by loop retiming. If the original energy consumption is large, the loop retiming technique will deliver less energy saving efficiency under the case of larger threshold values.

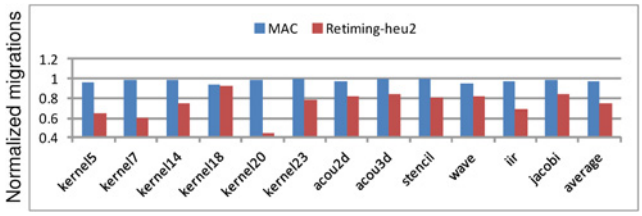



Fig. 13. Normalized migrations.

### *E. Comparison To MAC Method*

The MAC technique [16] is the closest work to the proposed technique. As aforementioned, the MAC is a migration-aware compilation approach by rearranging data layout in the memory, targeting migration reduction and energy efficiency improvement for hybrid cache. Its main concept is to place data with consecutively same access operations into the same memory blocks. The reduction of transition events leads to a reduction of migrations, thus improves the energy efficiency. We employ the MAC method to optimize our selected benchmarks and compare the results of migration reduction to our results with the *heu2* algorithm as shown in Fig. 13. The threshold values are configured as Case 2. The results with the two methods are both normalized to the default version.

We observe that, our results significantly outperform those with the MAC method in terms of migration reduction. This can be explained that, the MAC method is designed to deal with scalar variables which have interleaving accesses in the execution. With data replacement, the data with read transactions are placed in the same memory blocks while the ones with write transactions are placed in other different memory blocks. However, because the data in stencil loops with R/W dependencies have symmetric reads and writes, the MAC method cannot distinguish read intensive data and write intensive data. The MAC method has no effect on these data array elements, presenting low efficiency.

The proposed loop retiming can only handle data arrays in stencil loops, but cannot optimize scalar data. Since these two methods are orthogonal to each other, it is promising to further improve the dynamic energy by combining these two methods.

## VII. Conclusion

This paper presents the interesting observations that migrations in stencil loops are significant and often caused by interleaved access pattern. This access pattern often correlates closely to the R/W dependencies inside one memory block. We propose a loop retiming technique to reduce the interleaved accesses in a memory block so as to reduce migrations. An optimal method is proposed to obtain the optimal retiming solution by solving ILP problem sets. Two heuristic methods are proposed to derive the retiming vectors for the nodes one by one with low complexity. Experimental results show that the proposed techniques can significantly reduce the number of migrations, and improve energy efficiency for the hybrid cache. This retiming technique can also be effectively applied in the other asymmetric hybrid cache configurations, thus perhaps initiating further opportunities for new merging cache optimization techniques.

## REFERENCES


[1] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li, "Emerging non-volatile memories: Opportunities and challenges," in *Proc. 7th IEEE/ACM/IFIP CODES+ISSS*, 2011, pp. 325–334.

[2] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *Proc. ISCA*, 2009, pp. 34–45.

[3] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Write activity reduction on non-volatile main memories for embedded chip multiprocessors," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 3, pp. 77:1–77:27, 2013.

[4] J. Hu, C. J. Xue, W.-C. Tseng, Y. He, M. Qiu, and E. H.-M. Sha, "Reducing write activities on non-volatile memories in embedded CMPs via data migration and recomputation," in *Proc. 47th Design Autom. Conf.*, 2010, pp. 350–355.

[5] L. Jiang, B. Zhao, Y. Zhang, and J. Yang, "Constructing large and fast multi-level cell STT-MRAM-based cache for embedded processors," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 907–912.

[6] B. Zhao, J. Yang, Y. Zhang, Y. Chen, and H. Li, "Architecting a common-source-line array for bipolar non-volatile memory devices," in *Proc. Conf. Design, Autom. Test Eur.*, 2012, pp. 1451–1454.

[7] Z. Sun, H. Li, Y. Chen, and X. Wang, "Voltage driven nondestructive self-reference sensing scheme of spin-transfer torque memory," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 11, pp. 2020–2030, Nov. 2012.

[8] Y. Chen, X. Wang, H. Li, H. Xi, Y. Yan, and W. Zhu, "Design margin exploration of spin-transfer torque ram (STT-RAM) in scaled technologies," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 18, no. 12, pp. 1724–1734, Dec. 2010.

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

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

[11] L.-F. Chao and E.-M. Sha, "Scheduling data-flow graphs via retiming and unfolding," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 12, pp. 1259–1267, Dec. 1997.

[12] N. Passos and E.-M. Sha, "Achieving full parallelism using multidimensional retiming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 11, pp. 1150–1163, Nov. 1996.

[13] C. J. Xue, J. Hu, Z. Shao, and E. Sha, "Iterational retiming with partitioning: Loop scheduling with complete memory latency hiding," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 3, pp. 22:1–22:26, 2010.

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

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

[16] Q. Li, J. Li, L. Shi, C. J. Xue, and Y. He, "MAC: Migration-aware compilation for STT-RAM based hybrid cache in embedded systems," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, 2012, pp. 351–356.

[17] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Design exploration of hybrid caches with disparate memory technologies," *ACM Trans. Arch. Code Optimiz.*, vol. 7, no. 3, pp. 15:1–15:34, 2010.

[18] Q. Li, M. Zhao, C. J. Xue, and Y. He, "Compiler-assisted preferred caching for embedded systems with STT-RAM based hybrid cache," in *Proc. 13th ACM SIGPLAN/SIGBED LCTES*, 2012, pp. 109–118.

[19] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

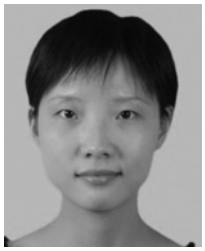
[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, 2004, pp. 75–86.

[21] *Livermore* (2013, May 15). [Online]. Available: http://www.netlib.org/benchmark/livermore/

[22] *Blitz++* (2013, May 15). [Online]. Available: http://blitzplus-pplus-p.sourcearchive.com/

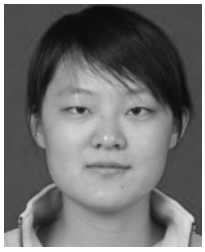
[23] *Pin* (2013, Dec. 14). [Online]. Available: http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool/

[24] 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. Microarch.*, 2007, pp. 3–14.

**Keni Qiu** received the B.S. degree from the Department of Information Technology, Central China Normal University, Wuhan, China, in 2001, and the M.S. degree from the Graduate University of Chinese Academy of Sciences, Beijing, China, in 2004, respectively. Since 2010, she has been pursuing Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong.

She was an Electronic Engineer with the Center of Space Science and Application Research, Chinese Academy of Sciences, Beijing, from 2004 to 2010. Her current research interests include embedded system and non-volatile memory optimizations.



**Mengying Zhao** received the B.S. degree from the Department of Computer Science and Technology of Shandong University, Jinan, China, in 2011. She is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong.

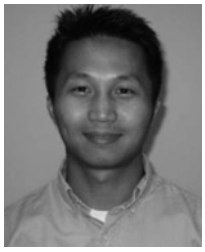
Her current research interests include embedded and real-time systems, architecture, and memory optimizations.

**Qingan Li** received the B.S. degree from the Computer School of Wuhan University, Wuhan, China, in 2008, where he is currently pursuing the Ph.D. degree. He is also with the Joint Program of City University of Hong Kong, Kowloon, Hong Kong.

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

**Chenchen Fu** received the B.S. degree from the Department of Computer Science and Technology, Shandong University, Jinan, China, in 2012, and is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong.

Her current research interests include scheduling algorithms for embedded systems and energy management.



**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, Kowloon, Hong Kong. His current research interests include memory and parallelism optimization for embedded systems, software/hardware codesign, real time systems, and computer security.