Compiler-Assisted Preferred Caching for Embedded Systems with STT-RAM based Hybrid Cache

Qingan Li
Computer School, Wuhan University
Department of Computer Science, City University of Hong Kong
ww345ww@gmail.com

Mengying Zhao
Chun Jason Xue
Department of Computer Science, City University of Hong Kong
my19900808@gmail.com
jasonxue@cityu.edu.hk

Yanxiang He
Computer School, Wuhan University
yxhe@whu.edu.cn

Abstract

As technology scales down, energy consumption is becoming a big problem for traditional SRAM-based cache hierarchies. The emerging Spin-Torque Transfer RAM (STT-RAM) is a promising replacement for large on-chip cache due to its ultra low leakage power and high storage density. However, write operations on STT-RAM suffer from considerably higher energy consumption and longer latency than SRAM. Hybrid cache consisting of both SRAM and STT-RAM has been proposed recently for both performance and energy efficiency. Most management strategies for hybrid caches employ migration-based techniques to dynamically move write-intensive data from STT-RAM to SRAM. These techniques lead to extra overheads. In this paper, we propose a compiler-assisted approach, preferred caching, to significantly reduce the migration overhead by giving migration-intensive memory blocks the preference for the SRAM part of the hybrid cache. Furthermore, a data assignment technique is proposed to improve the efficiency of preferred caching. The reduction of migration overhead can in turn improve the performance and energy efficiency of STT-RAM based hybrid cache. The experimental results show that, with the proposed techniques, on average, the number of migrations is reduced by 21.3%, the total latency is reduced by 8.0% and the total dynamic energy is reduced by 10.8%.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers; B.3.2 [*Design Styles*]: Cache memories

General Terms Design, Performance

Keywords Hybrid Cache, Data Assignment, Compiler

1. Introduction

As technology scales down, traditional SRAM-based caches encounter many challenges such as energy consumption and scalability. Recent advancements in memory technology present Spin-Torque Transfer RAM (STT-RAM) as a new candidate for building caches [6]. Compared to SRAM, STT-RAM has higher storage density, and much lower leakage power. However, write operations in STT-RAM have considerably longer latency and higher energy consumption than in SRAM. To take advantages of both SRAM and STT-RAM, hybrid cache architectures have been studied and evaluated in recent work [20] [21] [9] [12]. These studies show that caches built with multiple memory technologies have potential to outperform its counterpart of single technology. To efficiently utilize hybrid cache, migration-based techniques are commonly used to dynamically move write-intensive data from STT-RAM to SRAM. However, migrations require extra read and write operations for data movement and this extra overhead may degrade both the performance and energy efficiency of the hybrid cache. Embedded systems often have more stringent power and performance constraints. In this paper, we propose compiler-assisted techniques to improve the performance and energy efficiency for embedded systems with STT-RAM based hybrid cache by significantly reducing the migration overhead.

The overhead of migrations has not been well evaluated previously. In this paper, we have conducted a set of experiments and it is observed that the migration overhead is significant. We found that migrations correlate closely with the R/W transition events in access sequences (a R/W **transition event** represents a read followed by a write, or a write followed by a read, in the same memory block). The number of transition events can be used as an indicator for the number of migrations. It is also observed that most transition events in memory blocks come from access operations in the stack, rather than from the heap and global data area. In addition, for the stack, most transition events occur in a small group of memory blocks. With these observations, a compiler-assisted approach, preferred caching, is proposed in this paper to reduce the overhead of migrations by identifying these transition-intensive memory blocks and giving them the preference to be loaded into the SRAM part of the hybrid cache. The reduction of migration overhead will in turn improve both the performance and energy efficiency of STT-RAM based hybrid cache.

This paper makes the following contributions:

- Analyzes the migration overhead in STT-RAM based hybrid cache, and present several important observations.
- Proposes a compiler-assisted approach, preferred caching, to significantly reduce the migration overhead. The proposed approach identifies migration-intensive memory blocks and gives them the preference for the SRAM part of STT-RAM based hybrid cache. As a result, migrations for these preferred memory blocks are eliminated.
- Proposes a technique to improve the efficiency of the preferred caching approach by data assignment. Data assignment can

LCTES 2012 June 12–13, 2012, Beijing, China.
Copyright © 2012 ACM [978-1-4503-1212-7]. . . $10.00

change the access sequences in each memory block. After the new data assignment, migrations can concentrate in a smaller number of memory blocks. Giving these cache lines the preference in SRAM can further reduce migrations.

The rest of this paper is organized as follows. Section 2 presents the analysis of the migration overhead in STT-RAM based hybrid cache. Section 3 introduces the preferred caching approach. The efficiency of preferred caching could be improved by data assignment at compilation time. This technique is discussed in Section 4. The experimental results are presented in Section 5. The related works are introduced in Section 6. Finally, Section 7 concludes this paper.

2. Analysis of the migration overhead

This section introduces the analysis of the overhead of migrations in STT-RAM based hybrid caches. This paper focuses on embedded systems, in which the configuration of only one-level cache is often applied, such as Freescale e300 family [2], Renesas VR-series [4], and Cortex-R based MCUs [1]. For the evaluation in this section, we consider a one-level on-chip data hybrid cache, where the cache size is 32K bytes, the associativity is 4-way (one way for SRAM, and three ways for STT-RAM), and the cache-line size is 32 bytes. We defer the discussion of cache parameters to Section 5. The cache management strategy in [12] is implemented for this evaluation. We obtain four observations:

1. The overhead from migration based techniques in the STT-RAM based hybrid cache is significant.
2. The overhead of migrations correlates closely with the number of transition events in memory blocks.
3. Most transition events in memory blocks come from stack area, rather than from static area and heap area.
4. The number of transition events from the stack has an unbalanced distribution over different memory blocks.

2.1 Overhead of migrations

Migration based techniques are commonly proposed for non-volatile memory based hybrid caches [20] [21] [12]. For STT-RAM based hybrid caches, the SRAM part is preferable for write operations and the STT-RAM part is preferable for read operations. Considering the high probability that the program writes data to a specific group of cache lines repeatedly, cache lines in STT-RAM should be migrated to SRAM if they are frequently written to. This migration technique is first presented in [20] and then improved in [12]. The authors of [12] find that, when there is a need of migration, it is better to exchange data between two cache lines, rather than only migrate the data in STT-RAM to SRAM. In this paper, this kind of migration, triggered by write operations, is called *swap*. Furthermore, the same authors observe that there is also high probability that the program reads data from a specific group of cache lines repeatedly, and propose to migrate cache lines from SRAM into STT-RAM if they are read frequently. Then, the SRAM part can be set aside for write-intensive data. In this paper, this kind of migration, triggered by read operations, is called *migrate*. Both *swap* and *migrate* require extra read and write operations for data movement within the hybrid cache, and these extra overhead may degrade performance and energy efficiency of STT-RAM based hybrid caches.

Figure 1 shows the overhead of migrations for the selected benchmarks. In this figure, the migration consists of two parts: *swap* and *migrate*. It is found that, on average, eight migration operations are triggered per hundred memory accesses. It means that the migration overhead is significant. This overhead is detrimental to the system performance and energy consumption, and should be minimized.

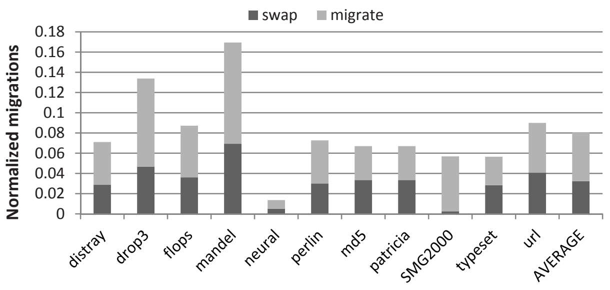


Figure 1. Normalized number of migrations (including both *swap* and *migrate*). The baseline is the number of memory accesses.

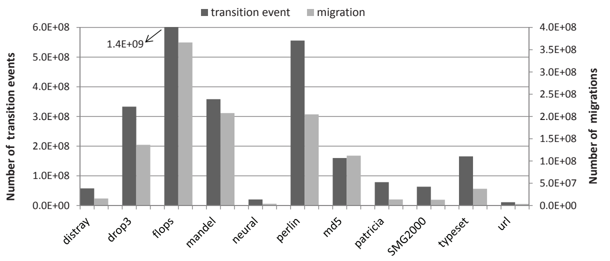


Figure 2. Correlation between transition events and migrations.

2.2 Correlation between migrations and transition events

It is observed that the migration overhead correlates closely with transition events in access sequences. A R/W **transition event** represents a read operation followed by a write, or a write operation followed by a read, in the same memory block. Generally, more transition events lead to more migrations. The reason is, a sequence of read operations followed by several write operations in a SRAM cache line triggers a *migrate*, and a sequence of write operations followed by several read operations in a STT-RAM cache line triggers a *swap*. Therefore, if too many transition events occur in memory blocks, the migration mechanism will be triggered frequently, in which situation, the benefits from migration will be offset by the migration overhead.

Figure 2 shows the correlation between the number of migrations and the number of transition events in memory blocks. It is found that transition-intensive memory blocks are often migration-intensive memory blocks. The number of migrations changes along with the number of transition events in memory blocks. For the selected benchmarks, the correlation coefficient between the number of migrations and the number of transition events is about 0.94. Therefore, the number of transition events, which is visible at compilation time, can be used as a good indicator to reduce the migration overhead.

2.3 Distribution of transition events over memory areas

Program data are stored in three kinds of memory areas: stack area, static area, and heap area. Local data, including local variables and compilation temporary variables are stored in stack area. Stack area also includes the space for register protection, the space for parameters, and the space for return addresses. Stack storage can efficiently deal with dynamic function invocation and provide space on demand. Static data, including global variables and static variables, are stored in global area. Heap area includes dynamically allocated space (*malloc* function in C, or *new* operator in C++). Heap storage

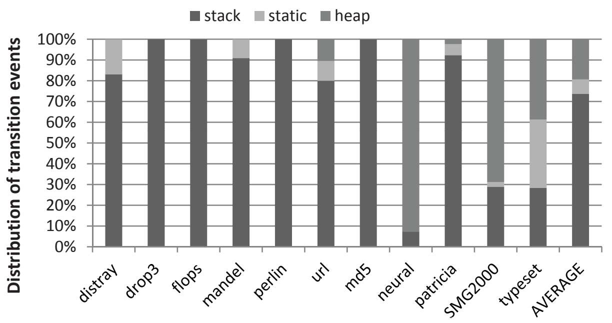


Figure 3. Distribution of transition events over memory area.

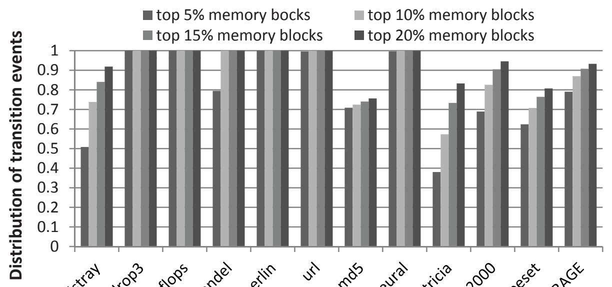


Figure 4. Distribution of transition events over memory blocks in the stack.

is commonly associated with the highest runtime cost among three kinds of storages, and thus is rarely applied in embedded systems because of the resource limitation.

Figure 3 shows the distribution of transition events over memory areas. It is found that for the selected benchmarks, on average, 74.0% of transition events come from the stack. Considering the correlation between migrations and transition events, it indicates that most migrations are from the stack. Therefore, this paper focuses on reducing the migration overhead originating from the stack.

2.4 Distribution of transition events over memory blocks

Figure 4 shows the distribution of transition events in the stack area over memory blocks. On average, 79.1% of transition events in the stack occur within the top 5% of memory blocks, and 93.3% of transition events in the stack occur within the top 20% of memory blocks. If we can statically identify these transition-intensive memory blocks at compilation time, and give them the preference for SRAM, then, no migrations are needed for these transition-intensive memory blocks. In other words, ideally we can eliminate about 80% of migrations by giving 5% of memory blocks the preference for SRAM. Furthermore, it is observed that the transition-intensive memory blocks are also write-intensive memory blocks. Therefore, giving transition-intensive memory blocks the preference for SRAM could also make more write accesses occur in SRAM, which is preferable to write operations.

These observations motivate the work proposed in this paper. A compilation method is proposed to identify the transition-intensive, essentially migration-intensive, memory blocks from the stack. By giving these memory blocks the preference for SRAM, it is possible to reduce the migration overhead. The reduction of migration overhead will in turn improve both performance and energy efficiency of the hybrid cache system. In this paper, a compiler-assisted approach, preferred caching, is proposed to significantly reduce the migration overhead. Furthermore, we present a technique to improve the efficiency of preferred caching by data assignment of stack objects at compilation time.

3. Preferred caching

As observed in Section 2.4, a small group of memory blocks often dominate the transition events in the stack. If we can correctly identify these transition-intensive memory blocks, migrations originating from these transition-intensive memory blocks could be eliminated by giving them the preference for SRAM. It is possible to identify transition-intensive memory blocks at compilation time using static profiling techniques. Furthermore, as observed in Section 2.2, the number of migrations correlates closely with the number of transition events. Therefore, by giving this small number of transition-intensive memory blocks the preference for SRAM, a large number of migrations can be eliminated.

This section introduces the proposed preferred caching approach. This compilation based approach consists of four steps. First, the information about which group of data objects will be loaded in the same memory block, is obtained. Second, the number of transition events within each memory block is computed. Third, the transition-intensive memory blocks are identified. Fourth, the transition-intensive memory blocks are given the preference for SRAM.

3.1 Identifying data objects belonging to the same memory block

At compilation time, we know each data object's offset relative to the stack base pointer, but cannot determine which group of data objects will be loaded into the same memory block during runtime. In other words, we need to know which group of data objects belong to the same memory block. If the stack base pointer for each function is aligned with the cache-line size, it will be easy to correctly identify which group of data objects belong to the same memory block according to the offset addresses relative to the stack base pointer. In this work, two tasks are carried out to align the stack base pointer with the cache-line size. First, extra instructions are inserted at the entry of the main function to align the stack base pointer of the main function with the cache-line size. Second, the stack size of each user function is expanded to be a multiple of the cache-line size. Assume that the cache-line size is *C*, and the original stack size of a function is *x*. After expansion, the new stack size for this function is: $\lceil \frac{x}{C} \rceil \times C$. After these two tasks, it is guaranteed that the stack base pointer of each user function invocation is aligned with the cache-line size.

After this alignment work, it is easy to identify data objects that belong to the same memory block. We only need to test whether the offsets of these two objects divided by the cache-line size are the same value. An example is shown in Figure 5, assuming that the cache-line size is 32-byte. After the alignment work, it is guaranteed that the stack base address (the value of the stack base pointer, *EBP*) is a multiple of 32. Therefore, in Figure 5, the address range $[EBP + 0, EBP + 31]$ constitutes the first block, and the address range $[EBP + 32, EBP + 63]$ constitutes the second block. Object c has an offset of 16, and belongs to the first memory block (16/32=0). Object f has an offset of 40, and object h has an offset of 56. As a result, f and h belong to the second memory block (40/32=1, 56/32=1).

3.2 Computing the number of transition events in memory blocks

The static profiling technique proposed in [22] can be used to estimate the transition events between data objects. This technique can estimate the execution frequency of each statement, each basic block and each control flow edge by exploring heuristics. With

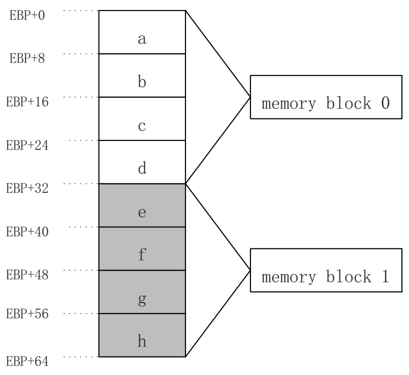


Figure 5. Mapping from stack offsets to memory blocks. Assume that the memory block size is of 32-byte. *EBP* is the stack base pointer for holding the address of the current stack frame.

this technique, the following information can be obtained at compilation time: the execution frequency of each block b, denoted as $\mathbf{Bfreq}(b)$, and the frequency from a block b to its succeeding block c, denoted as $\mathbf{Efreq}(b,c)$. By travelling the control flow graph (CFG) of each function, the transition events in each memory block could be estimated. The algorithm is illustrated in Algorithm 3.1. Within the loop from line 2 to line 19, each pair of consecutive data accesses is visited. In line 7 and line 16, each transition event is identified and the estimated occurrences of this transition event are collected. The method to collect transition events according to each pair of consecutive data accesses is detailed in Algorithm 3.2. For each pair of consecutive data accesses, if the corresponding two objects belong to the same memory block, a value is added to the number of transition events for this memory block. This added value is the execution frequency of a basic block b, if this pair of data accesses is within b; or the frequency of the edge e, if this pair of data accesses is across two basic blocks connected by e.

Algorithm 3.1 Estimating the number of transition events in each memory block.

```
Input:
    CFG: the CFG of a function
Output:
    transMap: a map storing the number of estimated transition events for
    each memory block
 1: initialize the weight of each element in transMap to be zero;
 2: for each basic block $b$ in the CFG do
 3:     // DA$(v,o)$ represents a data access: $v$ represents a data object, $o$
        represents the access type, read or write
 4:     // $da_0$ is used to record the previous data access
 5:     DA $da_0(v_0,o_0)$;
 6:     for each statements $s$ in basic block $b$ do
 7:         for each data access $da_1(v_1,o_1)$ in statement $s$ do
 8:             // a pair of consecutive accesses across basic blocks
 9:             if $da_1$ is the first data access within $b$ then
10:                 for the last data access $da_2(v_2,o_2)$ of each preceding
                    block $b_{pred}$ do
11:                     UpdateTrans(transMap, $da_1$, $da_2$, Efreq$(b_{pred},b)$ );
12:                 end for
13:                 // a pair of consecutive accesses within a basic block
14:             else
15:                 UpdateTrans(transMap, $da_1$, $da_2$, Bfreq$(b)$ );
16:             end if
17:         end for
18:     end for
19: end for
20: return true;
```

Algorithm 3.2 UpdateTrans: Updating transition events for each memory block according to each pair of consecutive data accesses.

```
Input:
    transMap: transition events for each memory block
    $da_1(v_1,o_1)$: the first data access
    $da_2(v_2,o_2)$: the second data access
    freq: the frequency value passed as parameter
Output:
    transMap: updated transition events for each memory block
 1: // skip transition events between the same object
 2: if $v_1$ == $v_2$ then
 3:     return 0;
 4: end if
 5: // skip non-transition events: read followed by a read, or write followed
    by a write
 6: if $o_1$ == $o_1$ then
 7:     return 0;
 8: end if
 9: int $b_1 \leftarrow v_1$.offset/CACHE_LINE_SIZE;
10: int $b_2 \leftarrow v_2$.offset/CACHE_LINE_SIZE;
11: // skip transition events across memory blocks
12: if $b_1$ != $b_2$ then
13:     return 0;
14: end if
15: // update transition events for memory blocks
16: transMap[$b_1$] += freq;
17: return true;
```

3.3 Choosing the transition-intensive memory blocks

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

3.4 Giving transition-intensive memory blocks the preference

There are many different ways that we can implement the preferred caching. A potential implementation method is illustrated in this section. Many Scum support functions of data pre-fetching and cache locking [2] [4] [1]. In these MCUs, the preferred caching can be implemented without any modification of the hardware. When the control flow enters a function, the transition-intensive memory blocks can be pre-fetched into the SRAM part of the STT-RAM based hybrid cache (if the target SRAM location is already locked, unlock it first), and then these cache lines can be locked using the cache locking function to avoid being evicted by cache replacement strategies and being migrated to STT-RAM. Note that in this paper the proposed preferred caching is a pre-emptive approach. It means, when the execution enters a function, its transition-intensive memory blocks can pre-empt the SRAM cache lines locked by previous functions (via the unlocking function). The detail is shown in Algorithm 3.3. These instructions for pre-fetching, cache locking and cache unlocking should be inserted at the entry of each function, and thus will be executed before any other instructions of this function.

Algorithm 3.3 CacheLock: Cache locking transition-intensive memory blocks into SRAM.

```
Input:
    blocks: the list of transition-intensive memory block of a function
 1: for each memory block $b$ in blocks do
 2:     // denote block $b$'s target location in SRAM as cache line $c$
 3:     // Step 1: cache unlocking
 4:     if $c$ is already locked then
 5:         insert instructions to unlock cache line $c$
 6:     end if
 7:     // Step 2: memory block pre-fetching
 8:     insert instructions to pre-fetch memory block $b$ into cache line $c$;
 9:     // Step 3: cache locking
10:     insert instructions to lock cache line $c$;
11: end for
12: return true;
```

4. Improvement of preferred caching by data assignment of stack objects

As stated in Section 2.2, the overhead of migrations is sensitive to the transition events in memory blocks. Changing the data assignment could change the transition events in memory blocks. If we could do the data assignment in a way that the distribution of write accesses is more concentrated, the effect of the proposed preferred caching would be enhanced. This is because, as the distribution of write accesses becomes more concentrated, more write accesses occur in a small group of memory blocks. As a result, identifying and giving this small group of memory blocks the preference for SRAM can eliminate more migrations. Furthermore, by giving this small group of memory blocks the preference for SRAM, more write accesses will occur in SRAM which is preferable to write accesses. This section introduces a stack data assignment method to improve the efficiency of the proposed preferred caching approach. The data assignment method consists of three steps:

1. Obtain the frequency of write accesses for each data object of stack.
2. Assign these data into a set of memory blocks according to their frequency of write accesses. The size of a memory block equals the cache-line size.
3. Finalize the data assignment.

In the first step, we can estimate the frequency of write accesses for each data object by either dynamic profiling or static profiling. The effectiveness of dynamic profiling is often sensitive to program input. In this work, we employ a static profiling technique [22] to estimate the frequency of write accesses for each data object. In the second step, we present a heuristic algorithm for the data assignment. The data objects are sorted in descending order by their write access frequencies, and then they are allocated by the sorted order. In the third step, the data assignment is finalized according to the previous data assignment. It is noteworthy to point out that the proposed algorithm does not handle objects of size greater than the cache-line size. These objects are left to be placed using the default method.

4.1 Obtaining write access frequency

A static profiling technique [22] is employed to obtain the execution frequency of each statement and each basic block. In Algorithm 4.1, from line 2 to line 9, each stack data access is visited, and the estimated frequency of each data accesses is collected according to the block frequency.

Algorithm 4.1 Obtaining write access frequency.

```
Input:
    CFG: the CFG of a function
Output:
    freqMap[]: an array with the write access frequency of each stack objects
 1: initialize the write access frequency of each stack objects to be zero;
 2: for each basic block $b$ in the CFG do
 3:     // $DA(v, o)$ represents a data access: $v$ represents a data object, $o$ represents the access type, read or write
 4:     for each statement $s$ in basic block $b$ do
 5:         for each stack data access $da_1(v_1, o_1)$ in statement $s$ do
 6:             freqMap[$v_1$] += Bfreq($b$);
 7:         end for
 8:     end for
 9: end for
10: return true;
```

4.2 Data assignment

We present a heuristic algorithm for the data assignment process. The detail of the proposed data assignment algorithm is shown in Algorithm 4.2. First, the data objects to be allocated are sorted by their write access frequency in descending order. Then, these data objects are assigned into memory blocks one by one. The size of each memory block is equal to the cache-line size. An example is presented in Figure 6. Note that whether a block b can hold an object d depends on the remaining space of memory block b, the size of data object d, and the required alignment of d.

Algorithm 4.2 Data assignment to memory blocks.

```
Input:
    blocks: a empty list of memory blocks
    freqMap: a map with the write access frequency of each stack objects
Output:
    blocks: a list of assigned memory blocks
 1: // Step 1: sort the data objects to be allocated
 2: sort the data objects by write access frequency in descending order;
 3: store the sorted objects into objects;
 4: // Step 2: allocate data objects into memory blocks
 5: while objects is not empty do
 6:     build a new memory block $b$ and add it into blocks;
 7:     for each data object $d$ of objects do
 8:         // consider both alignment and size of $d$
 9:         if $b$ can hold $d$ with regard to alignment requirement then
10:             remove $d$ from objects;
11:             allocate $d$ into $b$;
12:             update the remaining space of $b$;
13:             if $b$ is full then
14:                 break;
15:             end if
16:         end if
17:     end for
18: end while
19: // Step 3: allocate the unallocated data;
20: allocate the unallocated data using the default method
21: return $blocks$;
```

4.3 data assignment Finalization

After the data assignment process, a list of memory blocks is obtained. The offset of each data object internal to the memory block is also obtained. If the stack base pointer for each function is aligned with the cache line size, we can conduct the finalization of data assignment by mapping memory blocks into stack directly. This alignment work can be accomplished in the same way as mentioned in Section 3.1.

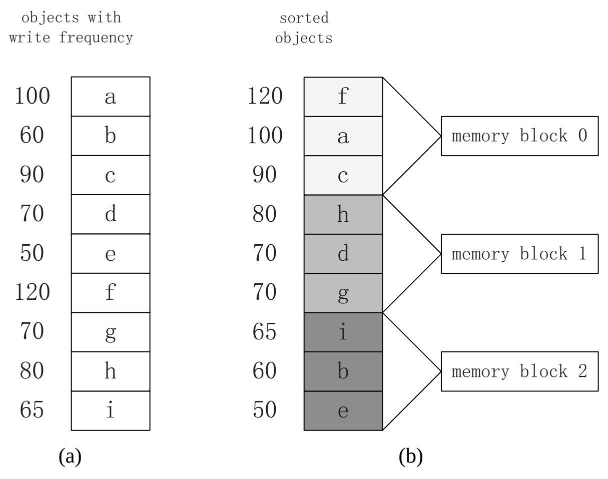


Figure 6. An example of data assignment. (a) The data objects to be assigned. The number before each object is the write access frequency for it. (b) The sorted data objects and the assignment results.

Table 1. Evaluated methods.

	Disable preferred caching	Enable preferred caching
Default data assignment	RG	PCG
Proposed data assignment	-	IPCG

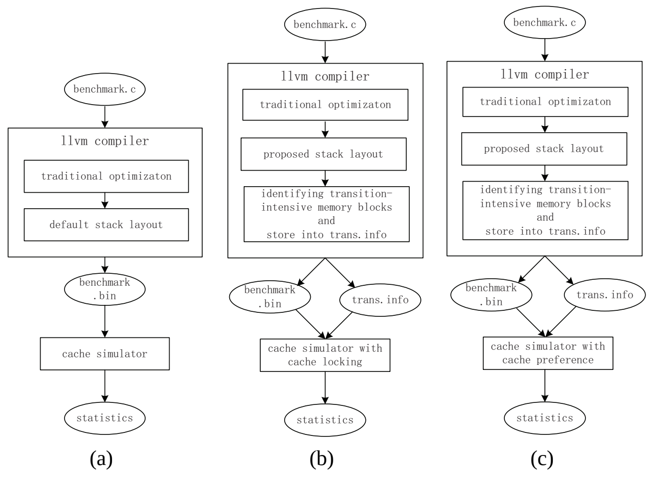


Figure 7. Experimental setup. (a) The RG group. (b) The PCG group. (c) The IPCG group.

5. Experiments

In this section, we first introduce the experimental setup, and then present the experimental results. Finally, a brief discussion about the runtime overhead of the proposed compilation approach is presented.

5.1 Experimental setup

The experimental setup is illustrated in Figure 7. The proposed compilation techniques are implemented based on LLVM [10], which is an open source compiler infrastructure. Three groups of executables shown in Table 1 are generated and the related statistics are collected. The first group, called the Reference Group (RG), is compiled by LLVM compiler using the default data assignment method (basically placing stack objects in the order of declaration). The second group, called the Preferred Caching Group (PCG), is compiled by LLVM compiler, giving transition-intensive memory blocks the preference to be loaded into SRAM cache lines. The third group, called the Improved Preferred Caching Group (IPCG), is compiled by LLVM compiler, combining data assignment with preferred caching transition-intensive memory blocks. For all of three groups, "O3" optimization in LLVM is enabled. The benchmarks as well as their input files are selected from the LLVM test suits, originated mainly from MiBench [7]. The characteristics of these benchmarks are shown in Table 2.

Table 2. Benchmarks characteristics.

Benchmark	Data Reads	Data Writes
distray	1.5E+08	7.7E+07
drop3	6.4E+08	3.8E+08
flops	3.3E+09	8.2E+08
mandel	7.0E+08	5.3E+08
neural	2.7E+08	1.1E+07
perlin	2.5E+09	2.6E+08
md5	1.4E+09	9.6E+07
patricia	1.4E+08	4.9E+07
SMG2000	1.8E+08	1.7E+08
typeset	2.5E+08	3.2E+07
url	3.5E+09	3.2E+08

Table 3. Architecture parameters.

Parameter	Value
processor	single core
hybrid data cache	32KB, 4-way, 32B cache-line size one way for SRAM (8KB), three way for STT-RAM (24KB) write allocation, write back SRAM access latency: 6 cycles SRAM access dynamic energy: 0.388 nJ STT-RAM read/write latency: 6/28 cycles STT-RAM read/write dynamic energy: 0.4/2.3 nJ
main memory	latency: 300 cycles

For the experimental evaluation, a Pin-based cache simulator is developed. Pin is a tool for the dynamic instrumentation of programs [15]. A cache simulator with the cache management strategy in [12] is implemented for the experimental evaluation. Note that the management strategy in [12] targets chip multiprocessors (CMPs), but this work focuses on micro-controllers (MCUs). Therefore, the simulator targets one-level data cache in single-core processors, and thus the inter-core migration strategy in [12] is not included. The target architecture is depicted in Table 3. The cache parameters and memory parameters are obtained from a modified group of CACTI [16].

5.2 Comparison with previous work

This section presents the comparison of the proposed techniques with the work proposed in [12]. There are several transactions that directly affect the total cost of memory access. Among these transactions, the most important ones are summarized in Table 4. In this table, the second column shows the atomic transactions related to each transaction in the first column, where MR represents STT-RAM read, MW represents STT-RAM write, SR represents SRAM read, SW represents SRAM write, MMR represents main memory read, and MMW represents main memory write. Our experiments evaluate the influences of the proposed algorithm on these transactions. Here, we choose 25 as the threshold N for identifying the transition-intensive cache lines to give the preference for SRAM. The experimental results are normalized to the baseline of RG.

The improvement of the proposed techniques on migrations are shown in Figure 8(a). Compared with RG, on average, the migrations are reduced by 18.9% in PCG and 20.6% in IPCG. The proposed techniques can also significantly reduce the number of write operations in STT-RAM, as shown in Figure 8(b). This reduction, on average 20.2% in PCG and 21.3% in IPCG, can significantly save cost since write operations in STT-RAM have longer latency and higher energy consumption than those in SRAM.

As discussed above, the proposed technique could reduce migrations, and reduce write accesses to STT-RAM. Therefore, it is promising that the proposed technique could improve the efficiency of the STT-RAM based hybrid cache. The total latency is evaluated using the cache and memory parameters shown in Table 3. Equation 1 is used to estimate the total latency, where MR_{num} represents the number of STT-RAM read transactions, MR_{cost} represents the cost (latency) per STT-RAM read transaction, MW_{num} represents the number of STT-RAM write transactions. The total dynamic energy consumption is estimated in a similar fashion. We only consider dynamic energy consumed by the cache, but not dynamic energy consumed by the main memory. As shown in Figure 8(c) and Figure 8(d), compared with RG, on average, the total latency is reduced by 7.2% in PCG and 8.0% in IPCG, and the total dynamic energy is reduced by 9.4% in PCG and 10.8% in IPCG.

$$\begin{aligned} total\_latency = & MR_{num} \cdot MR_{cost} + MW_{num} \cdot MW_{cost} \\ & + SR_{num} \cdot SR_{cost} + SW_{num} \cdot SW_{cost} \\ & + MMR_{num} \cdot MMR_{cost} \\ & + MMW_{num} \cdot MMW_{cost} \end{aligned} \tag{1}$$

5.3 The efficiency of preferred caching

Here we present discussions of the efficiency of the proposed techniques based on the results shown in Figure 8.

Reduction of migrations. Several factors may affect the efficiency of reducing migration overhead: the proportion of transition events occurring in the stack (see Figure 3), the distribution of transition events over memory blocks (see Figure 4), and the precision of the static profiling technique. Since the proposed approach focuses on the stack, the more transition events occur in the stack, the more migrations could be reduced. In addition, since the preferred caching approach relies on giving the transition-intensive memory blocks the preference for SRAM, the more concentrated the distribution of transition events is over memory blocks, the more migrations could be reduced. The precision of the static profiling determines the precision of the recognition of transition-intensive memory blocks, and thus affects the efficiency of preferred caching.

Let's review the reduction of migrations as shown in Figure 8(a).

For *distray*, *mandel*, *url*, and *SMG2000*, the results are reasonable. PCG brings good improvement, and IPCG improves even further.

For *drop3* and *perlin*, the results by PCG are extremely good, while IPCG performs poorly. Conversely, for *flops*, the results by IPCG are extremely good, while PCG performs poorly. It is further found that the distributions of stack transition events for these three benchmarks are extremely concentrated. There are two reasons. First, the static profiling technique is not accurate for these benchmarks. Second, with an inaccurate static profiling, the work of identifying transition-intensive memory blocks and giving them the preference for SRAM is not done well, and the side effects are amplified by a concentrated distribution of transition events over memory blocks.

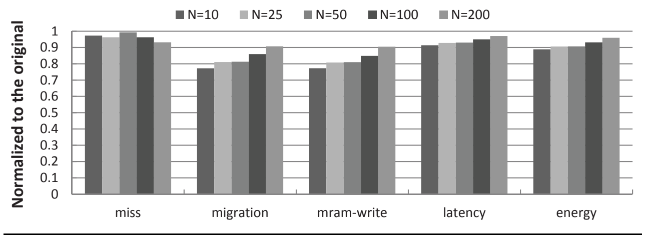


Figure 9. Choice of N. "mram-write" means the write accesses in STT-RAM. These data are average values over all selected benchmarks.

For *md5*, the reduction of migrations by PCG is not good. The reason is that the distribution of transition events in stack is not concentrated. As shown in Table 5, for this benchmark, no transition-intensive memory blocks are identified. However, by data assignment, the distribution is changed and the reduction is improved.

For *neural* and *typeset*, the improvement is marginal. This is because, the transition events occurring in the stack is less than 30% of the total transition events. Therefore, there are not so many migrations in the stack to be reduced.

For *patricia*, the result is not good. This is because, the distribution of stack transition events is not very concentrated, as the top 5% memory blocks take up less than 40% transition events. This makes the recognition of transition-intensive memory blocks difficult. As shown in Table 5, no memory block is identified as transition-intensive.

Reduction of total latency(or dynamic energy). Other than the factors affecting the efficiency of reducing migration overhead, there is one more factor affecting the efficiency of reducing the total latency (or dynamic energy): the number of migrations in the original group, RG (see Figure 1). Let's review the results of reducing the total latency (or dynamic energy) as shown in Figure 8(c) (or Figure 8(d)). For *SMG2000*, the reduction of migrations is significant, but the improvement on performance (or dynamic energy efficiency) is not so significant. This is because, there are not so many migrations for *SMG2000* in RG, which may degrade the efficiency of the proposed preferred caching technique.

5.4 The Choice of N

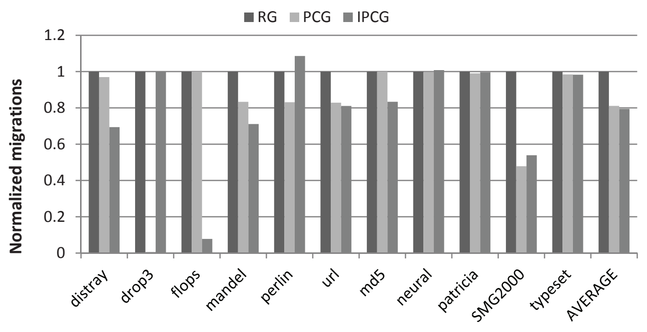
In this subsection, we discuss the choice of the threshold N which is used to identify the transition-intensive memory blocks. As illustrated in Figure 9, as N becomes larger, on average, the cache misses decrease, but migrations, and writes on STT-RAM increase. This phenomenon of double-edged effects can be explained as follows. As N becomes larger, fewer memory blocks will be identified as transition-intensive and fewer memory blocks will be given the preference for SRAM. The decrease of preferred memory blocks means more free memory blocks for replacement, thus the cache misses is consequently lower. In the meantime, the decrease of preferred memory blocks may reduce the benefit of the preferred caching approach, thus the migrations and writes on STT-RAM increase. However, in our experiments, the cache miss rate for each benchmark is always very small, so the final latency and final dynamic energy increase with N.

5.5 Overhead of proposed techniques

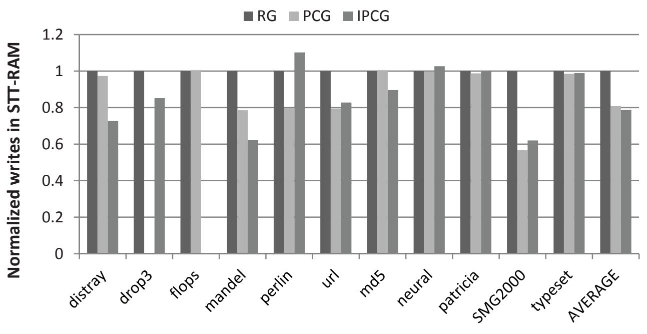
Overhead of preferred caching. The runtime overhead mainly comes from the implementation of preferred caching. As stated in Section 3.4, the preferred caching can be implemented by applying the pre-fetching and cache locking functions. Assume that the

Table 4. Important transactions.

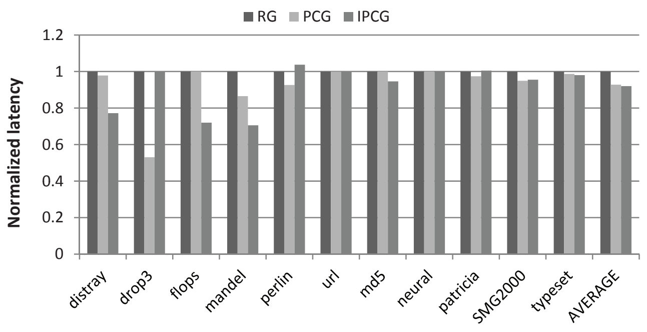
Transaction	Atomic transactions	Detail
Read Hit	MR/SR	read on STT-RAM/SRAM
Write Hit	MW/SW	write on STT-RAM/SRAM
Read Miss	MMR + MW + MR	fetch the targeted cache line from main memory into STT-RAM and read it
Write Miss	MMR + SW + SW	fetch the targeted cache line from main memory into SRAM and write it
Migrate	SR + MW	migrate a cache line from SRAM to STT-RAM
Swap	SR + SW + MR + MW	exchange two cache lines in SRAM and STT-RAM
Write Back	MR + MMW	write a cache line in STT-RAM back to main memory[1]



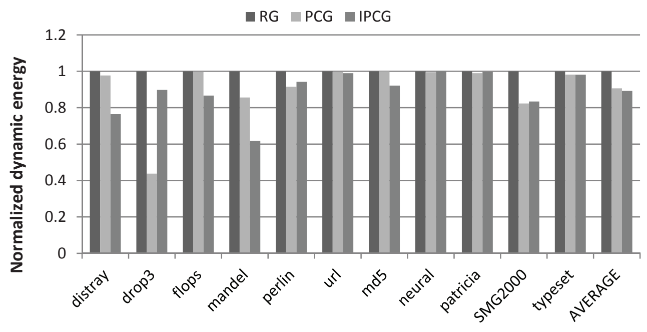
(a) Reduction of migrations.



(b) Reduction of write accesses in STT-RAM.



(c) Reduction of total latency.



(d) Reduction of total dynamic energy.

Figure 8. Comparison with RG. The results for PCG (Preferred Caching Group) and IPCG (Improved Preferred Caching Group) are normalized to results of RG (Reference Group).

number of transition-intensive memory blocks of a function f is f_{lines}, and f executes f_{exec} times. Then, for function f, the overhead of code size for preferred caching, is linear to f_{lines}; the runtime overhead is linear to $f_{lines} \cdot f_{exec}$. These values for PCG are collected in our experiments and shown in Table 5. From the second column to the fifth column, the value before '/' is the sum of f_{lines} for each benchmark, and the value after '/' is the sum of f_{exec} for each benchmark. A smaller N results in more overhead. It can be found that the runtime overhead is very small.

Other overheads. The proposed techniques may lead to the increase of stack size. For the preferred caching technique, the overhead lies in two aspects. First, there is space overhead for adjusting the initial stack address to be aligned with the cache-line size. This overhead is limited by the cache-line size. Second, there is space overhead for adjusting the stack size of each function to be a multiple of the cache-line size. This overhead is limited by *CACHE_LINE_SIZE* $\cdot$ *length*, where *length* is the number of functions in the longest function calling chain during runtime. The effects of data assignment on stack size are complicated, since the alignment requirements on data objects make the stack size varies with different data assignment methods and different stack base

Table 5. Overhead from preferred caching in PCG. For each benchmark, the value before '/' is the sum of f_{lines}, and the value after '/' is the sum of f_{exec}.

Benchmark	N=10	N=25	N=50	N=100
distray	4/1363924	1/1	1/1	1/1
drop3	2/40	2/40	2/40	1/20
flops	1/1	1/1	1/1	0/0
mandel	2/2	1/1	1/1	1/1
perlin	1/1	1/1	1/1	1/1
url	3/80900	2/80200	1/79800	1/79800
md5	0/0	0/0	0/0	0/0
neural	4/61	2/30	1/30	1/30
patricia	0/0	0/0	0/0	0/0
SMG2000	195/121385	153/106672	124/74761	98/71178
typeset	35/423179	26/315435	17/311450	10/77

[1] The management strategy for hybrid cache in [12] forces write back comes from STT-RAM.

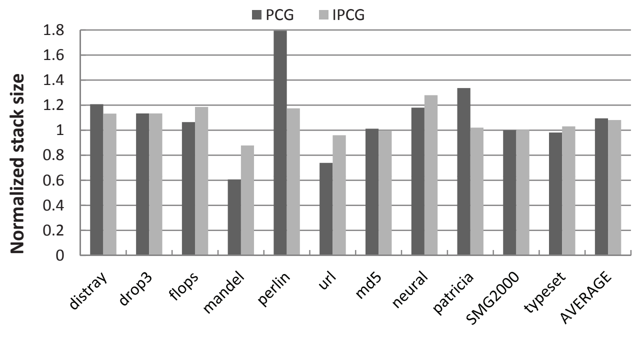


Figure 10. Comparison of stack size.

addresses. The comparison of stack size is shown in Figure 10. It shows, on average, the stack size is increased by less than 10%.

The proposed techniques may also lead to an increase of code size, since extra instructions are needed to adjust the initial stack address to be aligned with the cache-line size for the main function. However, no more than four instructions are needed for this task. Therefore, the space overhead as well as the runtime overhead from these instructions is negligible.

6. Related Works

Recently researchers have been attracted by several new memory technologies, including Embedded DRAM (EDRAM),Phase-change RAM (PRAM), and STT-RAM. These technologies are considered as potential candidates for building main memory [11], and cache [6]. Samsung has announced its PRAM based memory products which are used in mobile phones [3]. However, these technologies often suffer higher cost from write operations. Therefore, hybrid architectures are commonly employed. The hybrid main memory based on PCM and DRAM is presented in [18] [23] [24] [14]. The hybrid scratch pad memory is presented in [8]. The hybrid cache based on STT-RAM and SRAM is presented in [20]. Lots of work has been done to study management strategies for these hybrid architectures.

There are several studies architecting energy efficient STT-RAM based hybrid cache using hardware-based techniques. Sun et al. [20] propose hybrid caches consisting of SRAM and STT-RAM, and employed migration based policy to mitigate the drawbacks of STT-RAM. Wu et al. [21] evaluate two types of hybrid cache architectures (HCAs), including inter cache level HCA and intra cache level HCA, both of which employ cache line migration policy. Their experiments show that these HCAs can improve both energy efficiency and performance. More migration based policies are explored in [9] [12] to further improve the efficiency of STT-RAM based hybrid cache.

The compilation techniques for cache-aware data placement have been studied for a long time. A general framework for cache conscious data placement, including stack, global variables, and heap, is proposed in [5]. In this paper, the temporal relationship graph is employed to record the affinity relation between data objects and guide the placement of objects. A good survey on the problem of cache-aware data placement is presented in [17]. Recently, compiler-assisted caching techniques are also proposed in the context of multi-threaded architectures and multiprocessors [19] [13].

7. Conclusion

It is observed that the overhead of migrations for STT-RAM based hybrid caches is significant, and most migrations occur in a small group of memory blocks. With this observation, a compiler-assisted approach, preferred caching, is proposed in this paper to reduce the migration overhead by identifying these migration-intensive memory blocks and giving them the preference to be loaded into the SRAM part of the hybrid cache. Furthermore, a technique is presented to improve the efficiency of preferred caching by data assignment of stack objects. The experimental results show that the proposed techniques can improve both performance and energy efficiency.

Acknowledgments

This work was partially supported by the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 123609, CityU 123210, CityU 123811], and the National Natural Science Found of China [Project No. 61170022, 61173014, 61133005, 91118003].

References

[1] `http://www.arm.com/products/processors/cortex-r/index.php`.

[2] `http://cache.freescale.com/files/32bit/doc/ref_manual/e300coreRM.pdf`.

[3] `http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_NcPRAM.html`.

[4] `http://www2.renesas.com/micro/en/product/vr/legacy.html`.

[5] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 139–149, New York, NY, USA, 1998. ACM. ISBN 1-58113-107-0. doi: http://doi.acm.org/10.1145/291069.291036. URL `http://doi.acm.org/10.1145/291069.291036`.

[6] 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 *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 554–559, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: http://doi.acm.org/10.1145/1391469.1391610. URL `http://doi.acm.org/10.1145/1391469.1391610`.

[7] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3 – 14, dec. 2001. doi: 10.1109/WWC.2001.990739.

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

[9] A. Jadidi, M. Arjomand, and H. Sarbazi-Azad. High-endurance and performance-efficient design of hybrid cache architectures through adaptive line replacement. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, ISLPED '11, pages 79–84, Piscataway, NJ, USA, 2011. IEEE Press. ISBN 978-1-61284-660-6. URL `http://dl.acm.org/citation.cfm?id=2016802.2016827`.

[10] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL `http://dl.acm.org/citation.cfm?id=977395.977673`.

[11] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM. ISBN 978-

1-60558-526-0. doi: http://doi.acm.org/10.1145/1555754.1555758. URL `http://doi.acm.org/10.1145/1555754.1555758`.

[12] J. Li, C. Xue, and Y. Xu. Stt-ram based energy-efficiency hybrid cache for cmps. In *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on*, pages 31 –36, oct. 2011. doi: 10.1109/VLSISoC.2011.6081626.

[13] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 501–512, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: http://doi.acm.org/10.1145/1854273.1854335. URL `http://doi.acm.org/10.1145/1854273.1854335`.

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

[15] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065034. URL `http://doi.acm.org/10.1145/1065010.1065034`.

[16] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: http://dx.doi.org/10.1109/MICRO.2007.30. URL `http://dx.doi.org/10.1109/MICRO.2007.30`.

[17] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. *Nordic J. of Computing*, 12:275–307, June 2005. ISSN 1236-6064. URL `http://dl.acm.org/citation.cfm?id=1145884.1145889`.

[18] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: http://doi.acm.org/10.1145/1555754.1555760. URL `http://doi.acm.org/10.1145/1555754.1555760`.

[19] S. Sarkar and D. M. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 353–368, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-77559-5, 978-3-540-77559-1. URL `http://dl.acm.org/citation.cfm?id=1786054.1786087`.

[20] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. A novel architecture of the 3d stacked mram l2 cache for cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 239 –249, feb. 2009. doi: 10.1109/HPCA.2009.4798259.

[21] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid cache architecture with disparate memory technologies. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 34–45, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: http://doi.acm.org/10.1145/1555754.1555761. URL `http://doi.acm.org/10.1145/1555754.1555761`.

[22] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA, 1994. ACM. ISBN 0-89791-707-3. doi: http://doi.acm.org/10.1145/192724.192725. URL `http://doi.acm.org/10.1145/192724.192725`.

[23] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 101–112, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. doi: 10.1109/PACT.2009.30. URL `http://dl.acm.org/citation.cfm?id=1636712.1637751`.

[24] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: http://doi.acm.org/10.1145/1555754.1555759. URL `http://doi.acm.org/10.1145/1555754.1555759`.