DOI: 10.14529/jsfi190305

A Skewed Multi-banked Cache for Many-core Vector Processors*

Hikaru Takayashiki[1], ***Masayuki Sato***[1], ***Kazuhiko Komatsu***[1], ***Hiroaki Kobayashi***[1]

© The Authors 2019. This paper is published with open access at SuperFri.org

As the number of cores and the memory bandwidth have increased in a balanced fashion, modern vector processors achieve high sustained performances, especially in memory-intensive applications in the fields of science and engineering. However, it is difficult to significantly increase the off-chip memory bandwidth owing to the limitation of the number of input/output pins integrated on a single chip. Under the circumstances, modern vector processors have adopted a shared cache to realize a high sustained memory bandwidth. The shared cache can effectively reduce the pressure to the off-chip memory bandwidth by keeping reusable data that multiple vector cores require. However, as the number of vector cores sharing a cache increases, more different blocks requested from multiple cores simultaneously use the same set. As a result, conflict misses caused by these blocks degrade the performance.

In order to avoid increasing the conflict misses in the case of the increasing number of cores, this paper proposes a skewed cache for many-core vector processors. The skewed cache prevents the simultaneously requested blocks from being stored into the same set. This paper discusses how the most important two features of the skewed cache should be implemented in modern vector processors: hashing function and replacement policy. The proposed cache adopts the odd-multiplier displacement hashing for effective skewing and the static re-reference interval prediction policy for reasonable replacing. The evaluation results show that the proposed cache significantly improves the performance of a many-core vector processor by eliminating conflict misses.

Keywords: HPC, vector architecture, cache, skewed-associativity.

Introduction

Modern vector processors achieve high computing capability and high memory performance. Enhancing the architecture specialized for vector instructions with long vector lengths and the memory system focusing on high memory bandwidth, modern vector processors can achieve high sustained performances in scientific and engineering applications. The performance requirement of these applications is growing because these applications are eagerly developed for the demands of improving their accuracy and widening the applicable area. Thus, vector processors have been expected to provide higher performance.

In order to provide higher performance, modern vector processors have adopted two features different from the typical one. First, modern vector processors have increased their computing capability by increasing the number of vector cores, even though historically the vector processors have a single powerful core. As this trend may continue, many-core technology will play an essential role in driving computing capability growth in the future. Second, modern vector processors adopt multi-banked caches to keep the high sustained memory bandwidth. Since the further improvement of off-chip memory bandwidth is difficult due to the limitation of the number of input/output pins integrated on a single chip, a cache provides reusable data to vector cores at the high bandwidth.

As the number of vector cores increases, the off-chip memory readily becomes a bottleneck on memory-intensive applications in the future. In order for the vector processors to mitigate the gap between computing capability and off-chip memory performance, efficient utilization of the shared cache becomes an essential key factor. Since the shared cache can reuse the data among the

*The paper is recommended for publication by the Program Committee of the International Supercomputing Conference 2019 “HPC in Asia”.

[1]Tohoku University, Sendai, Japan

vector cores, unnecessary off-chip memory accesses can be reduced if the same data already exists on the shared cache. Thus, applications that run on the future vector processors are ought to be optimized to arrange the data on the shared cache as much as possible. Under the well-optimized applications, the number of cache hits will certainly increase if the number of vector cores sharing the data on the cache increases. Thus, less off-chip memory pressure can be realized by increasing the number of vector cores sharing data on the shared cache as much as possible.

First, this paper preliminarily evaluates the effect of the shared cache organizations on a many-core vector processor. A simple optimization is applied to an application so that more data can be shared as the number of vector cores increases. Thus, if the number of vector cores that share the cache architecturally increases, the more cache hit rate will be obtained. However, the preliminary evaluation clarified that a cache shared by many vector cores suffers from a lot of conflict misses. Although increasing the cache associativity is a common solution that can reduce the conflict misses, this solution brings a substantial overhead for a multi-banked cache.

Therefore, this paper proposes a skewed cache for many-core vector processors. The skewed cache is a cache that adopts skewed-associativity [13]. The skewed cache can suppress the number of conflict misses by preventing the simultaneous data requests of multiple vector cores from using the same cache set. This paper also discusses two features of the skewed cache: hashing functions, and replacement policies. The proposed cache adopts the odd-multiplier displacement hashing for effective skewing and the static re-reference interval prediction policy for reasonable replacing. In the evaluation, this paper evaluates the cache hit rates by using a stencil calculation kernel with varying the number of vector cores sharing a cache and its associativity. The evaluation results show that the proposed cache can eliminate the conflict misses and achieve nearly ideal hit rates in the case of the shared cache configurations.

The rest of the paper is organized as follows. Section 1 introduces a many-core vector processor assumed in this paper and indicates that the multi-banked cache suffers from conflict misses through the preliminary evaluation. Section 2 proposes a skewed multi-banked cache for many-core vector processor. Section 3 evaluates the proposal by simulation and discusses the results. The final section concludes this paper.

1. Challenges in Many-core Vector Processors

This section describes an organization of a many-core vector processor assumed in this paper. Then, various shared cache configurations of the many-core vector processor are preliminarily evaluated to show the impacts of conflict misses on the hit rates of the shared caches.

1.1. Many-core Vector Processors

1.1.1. Vector cores

Figure 1 shows the many-core vector processor assumed in this paper. One of the main parts of this processor is a vector core. The vector core is mainly divided into a scalar processing unit, a vector processing unit, and a load/store unit. The scalar processing unit fetches and decodes all the instructions and is responsible for the subsequent execution stages of scalar instructions.

The vector processing unit consists of vector operating units and vector registers. The vector operating units execute vector instructions, which can apply the arithmetic operations to vectorized data. A vector instruction can operate multiple elements at once. The number of elements

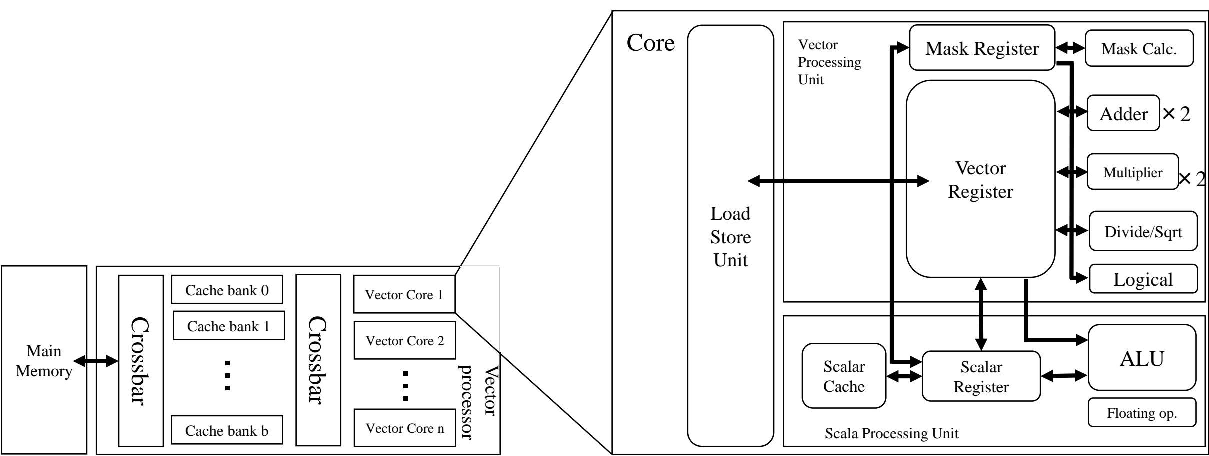


Figure 1. The overview of the many-core vector processor

that can be operated by one vector instruction is called *vector length*. For example, the maximum vector length is 256 in double-precision floating-point elements in the latest vector processor, SX-Aurora TSUBASA [8]. Compared to SIMD instructions of general-purpose processors, vector instructions can process large amount of data by one instruction.

The vector registers store operands used for executing vector instructions. The vector functional units can always be fed by the elements from the vector registers for calculations. These units consist of multiple arithmetic pipelines of addition, multiplication, division/square root, and logical calculation. Each unit can operate independently. Thus, it is possible to perform multiple types of operations in parallel. The vector mask register masks the execution results of a vector instruction. By using the vector mask register, loops including conditional branches are executed by using vector instructions.

The load/store unit generates memory addresses from vector load/store instructions and continuously sends memory access requests to the memory system. After all the data have arrived from the memory system, the data are transferred to the vector register immediately.

1.1.2. Multi-banked shared cache

With the improvement of the computing capability of vector cores, the demand for memory performance also increases to supply the data required by vector cores. However, the improvement of memory performance is behind in that of computing capability. Hence, cache memory is essential to fulfill the gap between them, and it is implemented in modern vector processors. For example, since the SX-9 or later vector processors of NEC SX series, an on-chip multi-banked cache memories are implemented inside the processor [5, 8]. This paper also assumes that the many-core vector processor includes a multi-banked cache memory.

If the number of vector cores increases in the future, it is considered that several numbers of cores are connected to one cache. Figure 2 shows an example where each cache is shared by M vector cores when the total number of vector cores is N. The following equation expresses the relationship between M and N

$$C = \frac{N}{M}, \tag{1}$$

where C is the number of caches, and M means the number of vector cores connected to one cache. Equation (1) indicates that the number of caches C varies depending on the M and N.

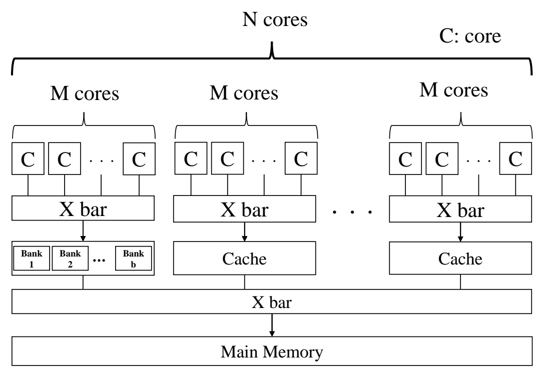


Figure 2. Example of M cores sharing the same cache in the N vector cores processor

```
 1: for z = 1, ...N_z − 1 do
 2:     for y = 1, ...N_y − 1 do
 3:         for x = 1, ...N_x − 1 do
 4:             b[z][y][x]=(a[z][y][x]+
 5:                     a[z][y][x-1]+a[z][y][x+1]+
 6:                     a[z][y-1][x]+a[z][y+1][x]+
 7:                     a[z-1][y][x]+a[z+1][y][x] )/7
 8:         end for
 9:     end for
10: end for
```

Figure 3. 3D 7-point stencil calculation

Thus, the number of cache banks in a cache, b, is expressed by the following equation.

$$b = \frac{B}{C} = \frac{BM}{N}, \qquad (2)$$

where B is the total number of banks in a many-core vector processor.

In this paper, N and B are constant because the various configurations of the shared cache are examined under the same computing capability and memory performance. Thus, the number of vector cores connected to the cache M only changes the number of banks per cache b according to Equation (2). Moreover, the capacity of each bank is fixed so that the number of banks per cache determines the cache capacity.

1.2. Conflict Misses on The Many-core Vector Processor

1.2.1. 3D 7-point stencil calculation

This paper examines various configurations of the shared cache using a representative memory-intensive computing kernel, the 3D 7-point stencil kernel. The stencil computations including this kernel occupy major roles in scientific and engineering simulation codes.

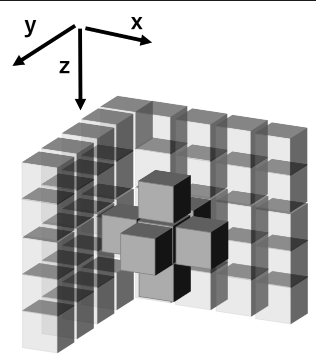
Figure 3 shows a pseudo-code for the 3D 7-point stencil kernel. This code derives arithmetic mean of a central element and its six neighboring elements along with x, y, and z axes. The calculation is repeated for all the elements in the 3D space. Thus, for each iteration of the inner-most loop of Fig. 3, a total of seven elements are required.

Figure 4(a) shows the elements used for one calculation in Fig. 3. The translucent elements represent the calculation space, and solid elements indicate the seven elements used for the calculation.

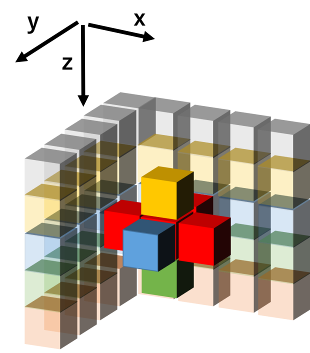
1.2.2. Stencil calculation with a shared cache

This paper assumes that the parallelization is performed based on the outermost loop regarding z-axis in Fig. 3. Therefore, each z-axis iteration of the loop is cyclically assigned to each core. The k-th iteration of the outermost loop is calculated by the core whose ID is (k mod N).

In the stencil calculation, each vector core accesses the central element and its neighboring elements. If other vector cores are already accessing these neighboring elements, the data might



(a) The elements used in one calculation



(b) The elements shared on the cache

Figure 4. 3D 7-point stencil kernel calculation

have already been placed in the cache. Thus, these elements can be reused on the cache, resulting in pressure reduction to off-chip memory.

Based on this parallelization, the theoretical cache hit rate of the 3D 7-point stencil calculation can be derived. Note that, in this paper, one layer of elements in an x-y plane cannot fit in the cache while one line of elements along with the y-axis can. First, the theoretical cache hit rate is calculated in the case where the vector core does not share the cache at all. Figure 4(b) shows which elements will hit on the cache. The translucent elements show the group of the elements that are calculated by one core. The solid blue element in Fig. 4(b) should miss irrespective of whether the cache is shared or not. The red elements: a[z][y][x], a[z][y][x-1], a[z][y][x+1], and a[z][y-1][x] should hit because the last iteration brings those elements in the cache irrespective of whether the cache is shared. The solid yellow and green elements cannot become hits because the one layer of the x-y plate cannot fit in the cache. Thus, in the case where the vector cores do not share a cache, the theoretical hit rate becomes 4/7 (57.14 %).

Next, the theoretical cache hit rate is calculated in the case where the vector cores share the cache with its neighboring cores, as shown in Fig. 2. The yellow and green elements can hit by sharing the cache because neighboring cores also bring the elements in the cache. Thus, a[z-1][y][x] and a[z+1][y][x] can hit additionally. However, since there is only one neighboring core for the cores which core numbers are 0 or $M-1$, either a[z-1][y][x] or a[z+1][y][x] always misses for these cores. From this observation, the 3D 7-point stencil calculation can be shared at most 2/7 (28.57 %) of elements by the shared cache.

Here, there are $7 \times M$ accesses by M cores when M iterations of the outermost loop are calculated in parallel. If the first core is calculating the outermost loop 1, the second core is calculating loop 2, ..., the M-th core is calculating loop M; the core in charge of loop 1 and the core in charge of loop M can reuse only 5 elements by the shared cache, and the other cores can reuse 6 elements by the shared cache. From this observation, for all the cores, the number of hits in one iteration is calculated as $6(M-2)+5\times 2$. Thus, the theoretical hit rate of 3D 7-point stencil calculation can be expressed as the following equation

$$H_7 = \frac{6(M-2)+5\times 2}{7M} = \frac{6}{7} - \frac{2}{7M}. \tag{3}$$

From Equation (3), it is possible to predict the hit rate of the stencil calculation theoretically from the number of cores sharing one cache M. It can be seen that, as the number of cores M increases, the cache hit rate monotonically increases, asymptotically approaching to $6/7$ (85.71 %).

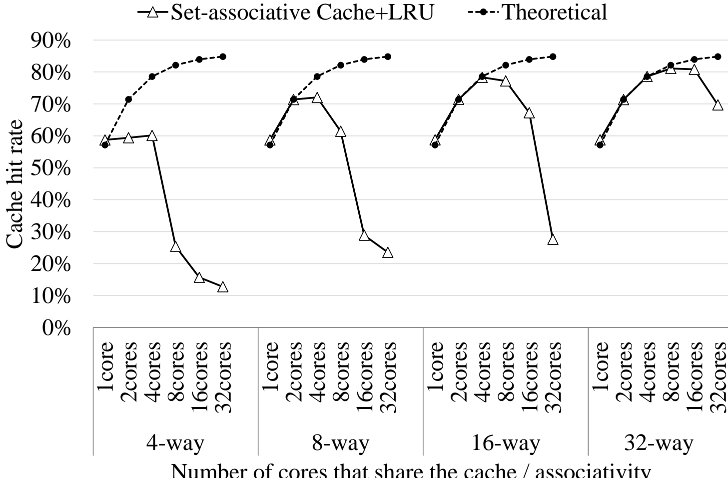


Figure 5. The cache hit rate when the number of cores sharing the same cache and the number of the associativity are changed

Overall, increasing the number of vector cores sharing a cache enables more vector cores to reuse the data on the shared cache. In the stencil calculation, the theoretical cache hit rate can increase if more cores share a cache.

1.3. Preliminary Evaluation

Equation (3) expresses the upper-bound of the cache hit rate because this equation only considers the reusability of the elements, and other effects are ignored. In order to confirm the model defined by Equation (3), a preliminary evaluation is conducted to investigate the hit rate in the many-core vector processor by varying the number of cores that share a cache. The stencil calculation is parallelized to share the data, as discussed in Section 1.2.2. Thus, the larger number of cores suggests the higher cache hit rate, as expected from Equation (3). Under this situation, various configurations of a many-core vector processor are evaluated. The number of cores sharing a cache is set to 1, 2, 4, 8, 16, and 32, and the cache associativity is set to 4, 8, 16, and 32. The other configurations correspond to the configuration described in Section 3.1.

Figure 5 shows the results of the preliminary evaluation. The vertical axis indicates the cache hit rate, and the horizontal axis indicates the number of cores sharing one cache and associativity. In Fig. 5, the theoretical cache hit rate increases as increases the number of cores sharing one cache. On the other hand, the cache hit rate of the set-associative cache with LRU replacement policy becomes significantly low in the case of a large number of cores sharing one cache and the low associativity. This is because, when the stencil calculation is executed in parallel, multiple cores simultaneously access the same set, resulting in conflict misses. Therefore, the number of conflict misses must be reduced to exploit the effect of sharing data among the vector cores by the shared cache.

1.4. Related Work

One of the methods to reduce conflict misses is to increase the associativity; however, it is challenging to realize the higher associativity because of cost, power consumption, and area overheads on the chip in large-capacity and multi-banked caches. Therefore, this paper discusses the other ways to eliminate conflict misses instead of increasing the associativity.

Some papers have tackled with the reduction in the number of conflict misses. Qureshi *et al.* [11] have proposed the V-Way cache, which includes the flexible tag mechanism, enables a global replacement to eliminate conflict misses. The V-Way cache has additional tag entries so that associativity of a set can vary depending on the demand to the set. Incidentally, the tag and data entries are associated with each other by indirection pointers. This mechanism enables to select victims flexibly from the global view of data, not limited to the same set. However, the V-Way cache requires a lot of additional hardware cost for the implementation.

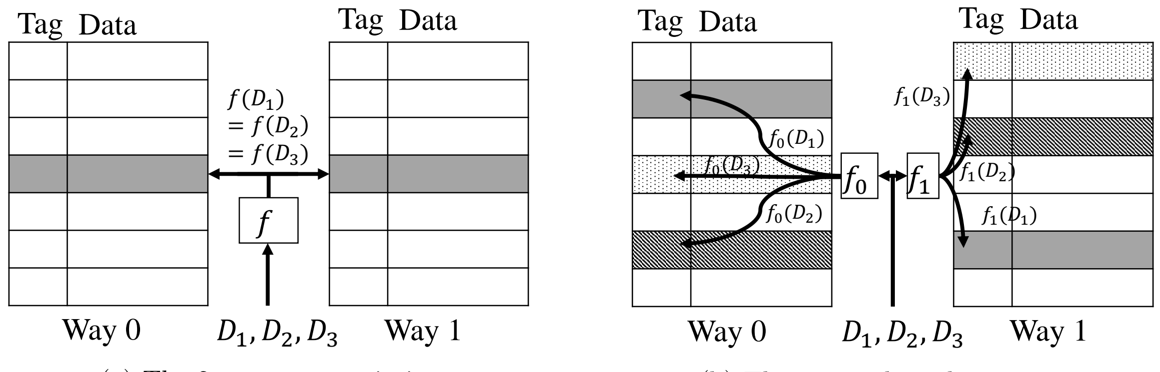
Sanchez *et al.* [12] have proposed the ZCache. The ZCache has focused on an insight that conflict misses occur by the lack of replacement candidates on an eviction. Hence, the ZCache selects several replacement candidates using multiple hashing functions for the same way on each miss. If a necessary block is about to be evicted, another useless block of another way would be evicted so that the necessary block can be relocated to and kept in the cache. However, the ZCache requires several array lookups to find a block in the cache, and additional data movement for the relocations. Consequently, these features cause too much cost for the multi-banked cache focusing on high bandwidth.

Although these proposals are effective in eliminating conflict misses, they require unignorable hardware and performance overheads. Moreover, the effect of these proposals is only evaluated for the scalar processors, not for the vector processors. Therefore, this paper focuses on a more straightforward way to reduce the number of conflict misses at low costs for the vector processors, looking back to the beginning of skewed-associativity.

There are several studies to clarify the usefulness of caches in vector processors. Musa *et al.* [10] have clarified that a four-core vector processor with one shared cache or two shared caches can improve the performance by 15-40% compared to the case without caches. This is because neighboring cores can reuse data by sharing the cache. Thus, the cache hit rate and the performance are improved. On the other hand, they have studied the effects of the shared caches by only up to 4 cores. The number of cores in modern vector processors is increasing, and this trend is expected to continue. Hence, a vector processor with more vector cores should be studied for the future. Additionally, they do not consider the cache associativity.

Egawa *et al.* [3, 4] have studied the shared cache up to 16 cores using real applications in a multi-core vector processor and clarified that increasing the number of cores and capacity of shared cache improves the hit rate. They have also clarified the improvement of the performance efficiency and the reduction in power consumption by the shared cache. Besides, they confirmed that an 8MB shared cache configuration is the most efficient for a 16-core vector processor. However, their studies are only for the number of cores and the cache capacity, not for the cache associativity. Additionally, the gap between the computing capability and the memory bandwidth has further widened after their study has done.

In order for the vector processors to obtain the high sustained performance even with increasing the number of vector cores, it is necessary to consider the cache associativity, as discussed in Section 1.3. Therefore, this paper studies the shared cache configurations for vector architectures in terms of the cache associativity and the number of cores sharing one cache, assuming that the number of vector cores significantly increases in the future.



(a) The 2-way set-associative (b) The 2-way skewed-associative

Figure 6. The difference of the 2-way set-associative and the 2-way skewed-associative

2. Skewed Multi-banked Cache for Many-core Vector Processors

Skewed-associativity [13] is an effective method to eliminate conflict misses without increasing the associativity. Hence, this paper proposes the skewed cache for the many-core vector processors in order for the shared cache to reduce the number of conflict misses on vector load/store data.

2.1. Skewed Cache

A skewed cache eliminates conflict misses by allocating a block to a set using the hashing function for each way. Figure 6 shows the difference between a 2-way set-associative cache and a 2-way skewed cache.

In Fig. 6(a), the set-associative cache allocates addresses D1, D2, and D3 to the same set. Thus, if the blocks of these addresses are stored in this order, the block of D1 inserted at first will be evicted by inserting the block of D3 due to the set conflict. In contrast, in Fig. 6(b), hashing function f_0 for way 0 and hashing function f_1 for way 1 individually generate different set indices so that these addresses are indexed to different sets. Since the block of D3 is stored to the different set where that of D1 is indexed to, it is possible to avoid evicting the block of D1 and avoid causing a conflict miss.

In order to obtain the high cache hit rate on the skewed cache, the hashing function and the replacement policy are essential. The hashing function intrinsically affects the ability of the skewed cache to avoid set conflicts. Thus, the hashing function should output non-biased values; otherwise, blocks are placed in the same set, causing conflict misses. Furthermore, it is desirable to easily create the various hashing functions based on a single rule so that they can produce different outputs for each way.

The replacement policy is also essential. One of the well-known problems with the skewed cache is that it is challenging to implement the Least Recently Used policy (LRU) on the skewed cache in the case of a higher associativity of three or more. This is because LRU generally determines an evicted block by using insertion order of blocks in the set. In the case of the set-associative cache, replacement candidates are always chosen from the same set, and an evicted block is selected from them. Thus, it is necessary to keep the insertion order within the sets. However, the skewed cache chooses replacement candidates from the different sets depending on the address of the block and the hashing function, so that the insertion order within the sets cannot determine whether the blocks recently used or not. If absolute timestamps are added to every block, LRU can be realized for the skewed cache. However, it requires an impractical hardware cost.

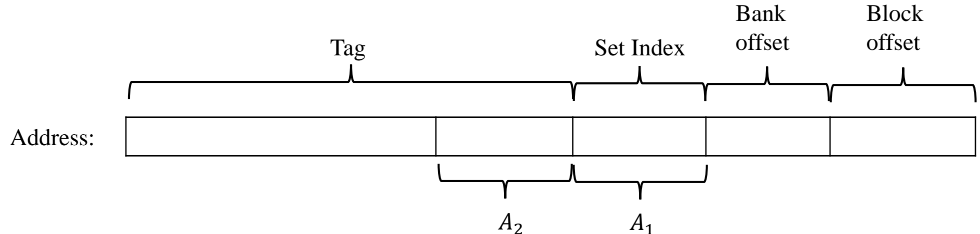


Figure 7. The bit field of hashing function of the given address

The remaining of this section discusses the implementations of the hashing function and the replacement policy that significantly affects the performance and implementation cost of the skewed cache.

2.2. Hashing Functions

2.2.1. XOR-based hashing function

An exclusive OR (XOR) based hashing function based on the XOR operations is used when the skewed-associativity is proposed [2, 13, 14]. The following equation calculates the set index.

$$index = \sigma^w(A_2) \oplus A_1 \bmod n_{set}, \tag{4}$$

in which $\oplus$ represents the bitwise exclusive OR operator. A_1 and A_2 denote fields of an address in Fig. 7. The bit lengths of A_1 and A_2 are $log_2(n_{set})$. σ^w represents a w-bit circular shift operation. Here, w generally represents the way ID for each way. For example, the hashing function for way 0 is $A_2 \oplus A_1 \bmod n_{set}$, and that for way 1 is $\sigma^1(A_2) \oplus A_1 \bmod n_{set}$.

The advantages of the XOR-based hashing function are simpleness for implementation, and this function satisfies two requirements for skewing. The blocks that may be mapped to the same set in a way are spread over other sets of the other ways. Moreover, two blocks with the same higher bits (A_2 or tag field in Fig. 7) cannot be mapped to the same set by the function.

However, the XOR-based hashing function is known to suffer from specific stride patterns called *pathological behavior* [7]. For example, if n_{set} is 16, A_1 is fixed, and A2 varies as a stride of 8, the XOR-based hashing function will generate the sequence of set indices 1,9,1,9,...,1,9. The same problem occurs even when w changes. Furthermore, if either A_1 or A_2 is 0 or 11..111, the outputs stay the same value despite any w. Thus, this paper examines another hashing function for the skewed cache.

2.2.2. Odd-multiplier displacement hashing function

This paper examines the Odd-Multiplier Displacement Hashing Function (oDisp) [7] as the alternative hashing function. the oDisp is known as a more uniform hashing function than the XOR-based hashing function. Thus, even in the case where access patterns contain strides of a specific length, the oDisp can be expected to reduce conflict misses further. The following equation expresses the set index

$$index = (o \times A_2 + A_1) \bmod n_{set}, \tag{5}$$

where o is an arbitrary odd number changed for each way.

```
Require: Candidates
Ensure: An evicted block
 1: if counter > (the number of blocks / 4) then
 2:     Reset RU of all blocks
 3:     Set counter 0
 4: else
 5:     Increment counter
 6: end if
 7: if Cache hits then
 8:     Set RU of the hit block to 1
 9: else if Cache misses then
10:     g1 :=Candidates.where(RU is 0)
11:     g2 :=Candidates.where(RU is 1 and clean)
12:     g3 :=Candidates.where(RU is 1 and dirty)
13:     if g1 is not empty then
14:         Return one randomly from g1
15:     else if g2 is not empty then
16:         Return one randomly from g2
17:     else if g3 is not empty then
18:         Return one randomly from g3
19:     end if
20: end if
```

Figure 8. The flow of NRUNRW policy

```
Require: Candidates
Ensure: An evicted block
 1: if Cache hits then
 2:     Set RRPV of the hit block to 0
 3: else if Cache misses then
 4:     while True do
 5:         for c in Candidates do
 6:             if c.RRPV is 3 then
 7:                 Return c
 8:             end if
 9:         end for
10:         Increment RRPVs of all candidates
11:     end while
12: end if
```

Figure 9. The flow of SRRIP policy

Another advantage of this hashing function is that the hardware cost is low. Equation (5) means that an arbitrary odd number multiplied by A_2 is added to A_1, and the remainder is calculated by the number of sets. Multiplication of an arbitrary odd number can be realized by one logical shifter and two adders instead of a multiplier. Furthermore, it is easy to change the output of the hashing function for each way by selecting different odd numbers for each way. Therefore, the index of the oDisp can be simplified by the following equation

$$index = ((A_2 \ll w) + A_2 + A_1) \bmod n_{set}, \tag{6}$$

where w represents the way number and $\ll$ represents a logical left-shift operation.

2.3. Replacement Policies

2.3.1. Not recently used not recently written

There are some studies about a replacement policy for the skewed cache, Seznec *et al.* [15] have proposed the Not Recently Used Not Recently Written policy (NRUNRW) based on the Not Recently Used (NRU). Figure 8 shows the flow of NRUNRW. This policy requires one bit per cache block as a Recently Used bit (RU) and selects one of the non-latest cache blocks as an evicted block. If the policy cannot determine the evicted block depending on the RUs of candidate blocks, the policy selects the evicted block from clean blocks among the candidate blocks. Furthermore, all RUs in the cache is reset every interval when the number of requests to the cache reaches one-fourth of the number of the total cache blocks.

The advantage of NRUNRW is that it requires a small hardware cost of only one bit per cache block. However, Seznec [14] pointed out that the performance of NRUNRW did not reach that of LRU because of the randomness by resetting the RUs. Therefore, this paper also examines an alternative replacement policy for the skewed cache.

2.3.2. Static Re-Reference Interval Prediction

In order to obtain the hit rate equivalent to LRU at a realistic hardware cost, the proposed skewed cache adopts the Static Re-Reference Interval Prediction policy (SRRIP) [6] as a replacement policy.

SRRIP is implemented as an extension of NRU and can achieve hit rates comparable to LRU with a simple replacement algorithm and low hardware cost. Fig. 9 shows the flow of SRRIP. SRRIP prepares an m-bit Re-Reference Prediction Value (RRPV) per cache block and predicts when the block will be re-referenced. SRRIP takes advantage of the fact that recently hit blocks have higher reusabilities than newly-inserted blocks. Since SRRIP lets hit blocks be hardly evicted, newly-inserted blocks can be evicted earlier than the recently hit block. Therefore, only recently hit blocks tend to remain in the cache, and other blocks are replaced.

SRRIP can be used for the skewed cache because the information used when replacing blocks, i.e. an RRPV for each block, can predict the re-reference interval of a block independent from the RRPVs of the other blocks in the same set. Thus, an evicted block can be determined even if replacement candidate blocks come from different sets, as in the case of the skewed cache.

3. Evaluation

3.1. Experimental Environment

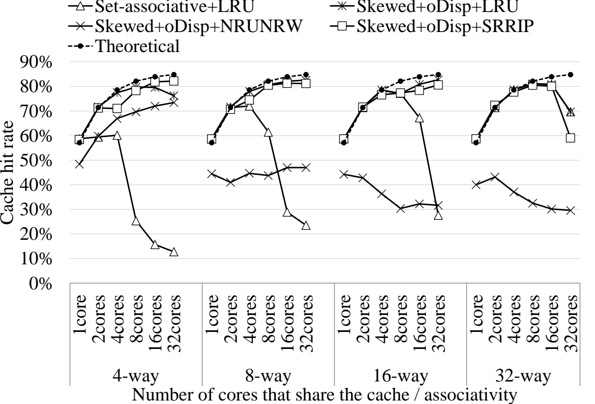
In order to confirm the effect of the skewed cache on the performance of many-core vector processors, we conduct experiments. The simulator of a many-core vector processor is developed based on the gem5 simulator [1], which is a general-purpose architecture simulator. The simulator uses an instruction trace data as an input, which is obtained by the vector supercomputer SX-ACE. Based on the trace data, it simulates the occupancy of hardware resources inside the processor and calculates the various performance metrics.

The simulation of the many-core vector processor is performed by implementing pseudo cores. The pseudo cores only issue requests to the memory system so that a simple implementation of pseudo cores enable to simulate many-core vector processors at high speed. In the stencil calculation targeted in this paper, it is assumed that parallelization is performed in the outermost loop of Fig. 3 discussed in Section 1.2.2. Therefore, the widths specified for the pseudo cores correspond to the amount of data for one z-iteration. The space of the calculation is set to 2048x2048x512.

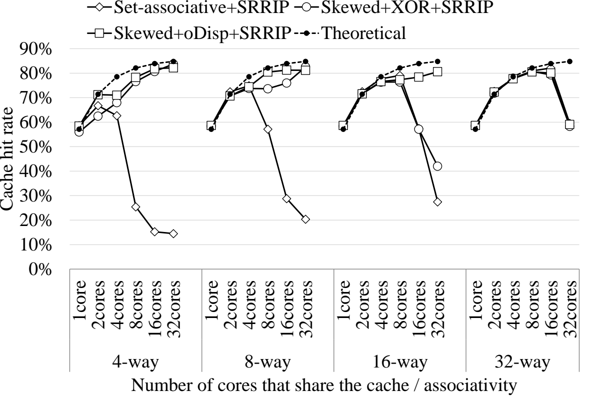
Table 1 shows the system configurations of the many-core vector processor. The total number of cores is set to 32 based on the trend in the future many-vector core processor, and the configuration of the core is based on that of NEC SX-ACE [5]. In addition, this paper assumes that the Bytes/Flop (B/F) value of the system used for the evaluation is set to 0.125. This is because the trend in the gap between computing capability and memory performance has widened, and the B/F value is diminishing. In fact, the B/F values of SX-ACE and SX-Aurora TSUBASA are 1.0 and 0.5, respectively. The newer generation vector processors ought to have lower B/F values.

Table 1. Configurations of the simulation

Base architecture	NEC SX-ACE
Total number of core	32
Number of cores sharing a same cache	1, 2, 4, 8, 16, 32
Main memory bandwidth	256GB/s
Total cache size (size per bank)	32MB (128KB)
Associativity	4, 8, 16, 32
Cache block size	128Bytes
MSHR (Target) [9]	8 (8)
System B/F	0.125 B/F



(a) Cache hit rate result with the same hashing function, oDisp, except set-associative cache



(b) Cache hit rate result with the same replacement policy, SRRIP

Figure 10. Cache hit rate results

In the evaluation, several parameters affect the performance of the multi-banked cache memory. The associativity is set to 4, 8, 16, and 32, and the number of cores sharing one cache is set to 1, 2, 4, 8, 16, and 32. The total capacity and the total number of banks are set to 32MB and 256 banks, respectively. The total capacity is fixed during the evaluation to focus on only the effect of the shared cache, and the total number of banks is also fixed to keep the bandwidth the same in all the configurations.

In addition to the proposed skewed cache, we evaluate the conventional set-associative cache to compare the results. On the proposed cache, NRUNRW, LRU, and SRRIP policies are used as replacement policies. This paper sets two bits to the RRPV for SRRIP in the evaluation, which can achieve the performance comparable to LRU [6]. Since the implementation of LRU in the realistic hardware cost for the skewed cache is difficult, LRU is implemented as the insertion order of blocks is judged by the absolute timestamps given to blocks.

3.2. Evaluation Results and Discussion

3.2.1. Cache hit rate

Figure 10 show the cache hit rates of the conventional cache and the skewed cache with various replacement policies and hashing functions. In the figures, the vertical axis represents the cache

hit rate, and the horizontal axis represents the number of cores sharing a cache and associativity of the cache.

Figure 10(a) shows the cache hit rate with various replacement policies. In order to discuss the difference among the replacement policies, we use the odd-multiplier displacement hashing as the hashing function. The hit rates of the proposed skewed cache with SRRIP are very close to the theoretical hit rates for each configuration. SRRIP follows theoretical values only when the associativity is 4-way or 8-way, although the hit rates of the high associativities are slightly lower than those of LRU. Compared to the set-associative cache when the cache is 4-way and 8-way, the number of hits is improved by a maximum 70% and 60%, respectively. In the case where the 32 cores share a 16-way cache, the number of hits increases by about 50%.

In contrast, the cache hit rate drastically decrease with the low associativity in the set-associative cache. Particularly, when the associativity is 4-way, 8-way or even 16-way, and the number of cores sharing a cache is large, the gap between set-associative cache and theoretical hit rates becomes apparent. This is because the stencil calculation consists of relatively regular memory access patterns. In the evaluation, the stencil calculation is parallelized, as discussed in Section 1.2.2. Every core issues memory access requests of the z-iteration assigned to each core. However, a difference in the requests can be distinguished mainly by the tag field of their addresses. Thus, these requests tend to be indexed to the same set in the set-associative cache, which results in conflict misses. On the other hand, the skewed cache could avoid conflict misses since it uses a broader range of an address field including a part of the tag field so that the skewed cache can distinguish the difference of the z-iteration on the address to determine the set.

As a replacement policy of the skewed cache, the hit rate of LRU achieves almost equal to the theoretical value. When LRU is applied, the skewed cache can maintain a higher hit rate than the set-associative cache or a high hit rate equal to the skewed cache with SRRIP. This is because the skewed cache with LRU is based on the timestamp so that blocks can be ideally replaced. Although its implementation requires impractical hardware costs, it is possible to eliminate more conflict misses than SRRIP regardless of the associativity. Note that NRUNRW has a low hit rate in all cases. This is because all the RUs are reset at each fixed access interval so that an evicted block can be randomly selected every interval. It causes that the necessary blocks are suffering from random eviction.

When the number of cores sharing one cache is 16 cores or more, and the associativity is 32-way, the cache hit rate decreases on both SRRIP and LRU. Two reasons can be considered. The first reason is that the number of sets per way decreases if the associativity is too high. Therefore, the possibility to select the same set many times becomes high, resulting in increasing the number of conflict misses. The second reason is due to the implementation matter of the oDisp hashing functions. In the skewed cache, the oDisp hashing function is implemented, as shown in Equation (6). When the number of ways is larger than the number of bits of A_2, the shifted A_2 does not affect the calculation of the index on the larger way numbers than the bitfield length of A_2. Therefore, these cache hit rates could become lower than those of the theoretical value. Note that the miss rates of SRRIP are slightly lower than those of LRU. When the associativity is very high, SRRIP meets many replacement candidates with the same RRPVs at once and becomes closer to a random replacement, which causes the increase in the number of misses.

Figure 10(b) shows the hit ratio of the skewed cache where a hashing function is changed. In order to discuss the difference among the hashing functions, we use SRRIP for the replacement policy of the skewed cache and the set-associative cache. From Fig. 10(b), it is observed that both

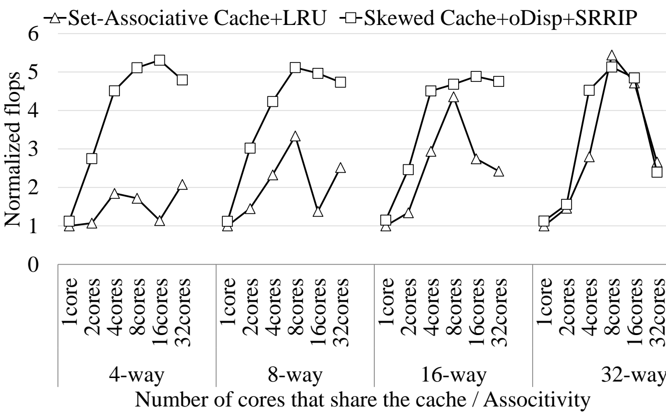


Figure 11. Performance comparison of the conventional set-associative cache and the proposed skewed cache

hashing functions can obtain a high cache hit rate across almost all cases. However, in the case where the associativity and the number of cores sharing one cache are extremely large, the XOR-based hashing function decreases the cache hit rate. Two reasons can be considered. The first reason is that the skewed cache with the XOR-based hashing function sometimes suffers from the specific access pattern called *pathological behavior*, as mentioned in Section 2.2.1. It is considered that, because the memory addresses requested from the vector cores coincidentally meet these kinds of specific stride pattern, the cache hit rate degradation is observed, specifically in the case of 16-way cache shared by 16 cores and 32 cores. The second reason is due to the implementation matter of the XOR-based hashing functions. When the number of ways is larger than the number of bits of A_2 on the XOR-based hashing function, the bit rotation is circulated, and the same index is calculated again on the larger way numbers. This leads to the cache hit rate degradation. In contrast, the oDisp can take care of this kind of access patterns. Therefore, it can obtain a stable high cache hit rate by the oDisp. Note that the set-associative cache with SRRIP suffers from lowering the cache hit rate due to the same reason of the set-associative cache with LRU in Fig. 10(a).

3.2.2. Performance

Figure 11 shows the performance comparison of the set-associative cache with the proposed skewed cache. The set-associative cache is based on LRU, and the skewed cache with oDisp & SRRIP is the proposed cache that can achieve a high hit rate at a reasonable implementation cost. In Fig. 11, the vertical axis represents the performance normalized by the case where each core privately owns a set-associative cache. The horizontal axis shows the number of cores sharing a cache and the associativity. Figure 11 shows that the proposed cache outperforms the set-associative cache when the associativity is low. Notably, in the case where 16 cores share a 4-way cache, the skewed cache obtains a six times higher performance than the set-associative cache. On the other hand, if there is sufficient associativity, there are no significant differences in performances between the skewed cache and the set-associative cache. This is because, if there is sufficient associativity, fewer conflict misses occur in both cases.

In Fig. 11, there are typical cases where one cache shared by 16 cores, and its performance is higher than that of the case shared by 32 cores, although the cache hit rate in 32 cores is higher than that in 16 cores. It is due to cache bank conflicts. Since the cache configuration of the many-core vector processor assumes a multi-banked configuration in this paper, memory access requests

go to each cache bank according to the memory address. Therefore, when many cores share one cache, the possibility of accessing the same cache bank is increasing. Additionally, in this cache configuration, the write buffer size and MSHR per bank are only 16 and 8, respectively. Therefore, in the case where 32 cores share a cache, their requests are concentrated on some bank due to bank conflicts, and the shortages of the write buffer or MSHR cause the performance degradation.

Overall, the skewed cache can realize the better performance than the conventional set-associative cache, especially when the associativity is low. This is because the skewed cache can successfully eliminate conflict misses. It is clarified that SRRIP can bring almost ideal cache hit rate to the skewed cache at reasonable implementation costs. Moreover, the oDisp hashing function avoids performance degradation from conflict misses.

Conclusions

This paper proposes a skewed multi-banked cache for many-core vector processors. The skewed cache can prevent simultaneously requested blocks from using the same cache set. The data block is stored for a cache set by using a hashed value of theÂăblock address. This paper discusses how the most important two features of the skewed cache should be implemented: the hashing functions and the replacement policies. This paper evaluates three replacement policies, LRU, NRUNRW, and SRRIP, and two hashing functions, XOR-based and oDisp for the skewed cache.

The evaluation results show that the skewed cache with SRRIP and oDisp can increase the number of hits by up to 70% and marks the closest results to those of the theoretical upper bound of the hit rate of the shared cache. From the individual evaluations of hashing functions and replacement policies, SRRIP can obtain the highest cache hit rate at low hardware cost, and oDisp can solve the problem of the XOR-based hashing function. The evaluation results also show that the skewed cache can realize a six times higher performance than the conventional set-associative cache on the stencil calculation. As future work, the skewed cache should be evaluated with more various real applications developed for modern vector processors.

Acknowledgements

This work is partially supported by MEXT Next Generation High-Performance Computing Infrastructures and Applications R&D Program, entitled ”R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications” and Grants-in-Aid for Early-Career Scientists No. 19K20232. The experimental results in this research were partially obtained by supercomputing resources at Cyberscience Center, Tohoku University.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., et al.: The Gem5 Simulator. SIGARCH Comput. Archit. News 39(2), 1–7 (2011), DOI: 10.1145/2024716.2024718

2. Bodin, F., Seznec, A.: Skewed associativity improves program performance and enhances predictability. IEEE Transactions on Computers 46(5), 530–544 (1997), DOI: 10.1109/12.589219

3. Egawa, R., Funaya, Y., Nagaoka, R., Endo, Y., Musa, A., Takizawa, H., Kobayashi, H.: Effects of 3-D stacked vector cache on energy consumption. In: 2011 IEEE Int. 3D Systems Integration Conf. (3DIC), 2011 IEEE Int. pp. 1–6 (2012), DOI: 10.1109/3DIC.2012.6263026

4. Egawa, R., Funaya, Y., Nagaoka, R., Musa, A., Takizawa, H., Kobayashi, H.: Design and early evaluation of a 3-D die stacked chip multi-vector processor. In: 2010 IEEE International 3D Systems Integration Conference (3DIC). pp. 1–8 (2010), DOI: 10.1109/3DIC.2010.5751448

5. Egawa, R., Komatsu, K., Momose, S., Isobe, Y., Musa, A., Takizawa, H., Kobayashi, H.: Potential of a Modern Vector Supercomputer for Practical Applications: Performance Evaluation of SX-ACE. J. Supercomput. 73(9), 3948–3976 (2017), DOI: 10.1007/s11227-017-1993-y

6. Jaleel, A., Theobald, K.B., Steely, Jr., S.C., Emer, J.: High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). SIGARCH Comput. Archit. News 38(3), 60–71 (2010), DOI: 10.1145/1816038.1815971

7. Kharbutli, M., Solihin, Y., Lee, J.: Eliminating Conflict Misses Using Prime Number-Based Cache Indexing. IEEE Trans. Comput. 54(5), 573–586 (2005), DOI: 10.1109/TC.2005.79

8. Komatsu, K., Momose, S., Isobe, Y., Watanabe, O., Musa, A., Yokokawa, M., Aoyama, T., Sato, M., Kobayashi, H.: Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 685–696 (2018), DOI: 10.1109/SC.2018.00057

9. Kroft, D.: Lockup-free Instruction Fetch/Prefetch Cache Organization. In: Proceedings of the 8th Annual Symposium on Computer Architecture. pp. 81–87. ISCA '81, IEEE Computer Society Press, Los Alamitos, CA, USA (1981), http://dl.acm.org/citation.cfm?id=800052.801868

10. Musa, A., Sato, Y., Soga, T., Okabe, K., Egawa, R., Takizawa, H., Kobayashi, H.: A Shared Cache for a Chip Multi Vector Processor. In: Proceedings of the 9th Workshop on MEmory Performance: DEaling with Applications, Systems and Architecture. pp. 24–29. MEDEA '08, ACM, New York, NY, USA (2008), DOI: 10.1145/1509084.1509088

11. Qureshi, M.K., Thompson, D., Patt, Y.N.: The V-Way cache: demand-based associativity via global replacement. In: 32nd International Symposium on Computer Architecture (ISCA'05). pp. 544–555 (2005), DOI: 10.1109/ISCA.2005.52

12. Sanchez, D., Kozyrakis, C.: The ZCache: Decoupling Ways and Associativity. In: 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 187–198 (2010), DOI: 10.1109/MICRO.2010.20

13. Seznec, A.: A Case for Two-way Skewed-associative Caches. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 169–178. ISCA '93, ACM, New York, NY, USA (1993), DOI: 10.1145/165123.165152

14. Seznec, A.: A New Case for Skewed-Associativity. Research Report RR-3208, INRIA (1997), https://hal.inria.fr/inria-00073481

15. Seznec, A., Bodin, F.: Skewed-associative caches. In: Bode, A., Reeve, M., Wolf, G. (eds.) PARLE '93 Parallel Architectures and Languages Europe. pp. 305–316. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)