Multi-dimensional Global Store Statement Aggregation

Yudai Yamagishi
Dept. of Computer Science
Takushoku University
Tokyo, Japan
sumikawa.lab@gmail.com

Yasunobu Sumikawa
Dept. of Computer Science
Takushoku University
Tokyo, Japan
ysumikaw@cs.takushoku-u.ac.jp

***Abstract*—Cache memory improves performance through data reuse. In multi-dimensional arrays, elements with matching higher-order indexes are placed close in memory, making consecutive accesses to such elements beneficial for cache reuse. We propose multi-dimensional global store statement aggregation (MDGSA), which reorders store statements to maximize consecutive accesses with matching higher-order indexes. MDGSA traverses the control flow graph in reverse topological order and moves each store statement to the earliest valid position. Implemented in the COINS compiler, MDGSA reduced cache write misses compared with existing code motion algorithms.**

***Index Terms*—code optimization, cache memory, code motion, multi-dimensional array access**

I. Introduction

Many contemporary computers use cache memory to reduce the frequency of accessing the main memory. If a program writes data to main memory and data exists at the same address in the cache memory, it is called a *write hit*; otherwise, it is a called *write miss*. Upon encountering a write miss, a procedure is initiated to reconcile the data consistency between the cache and main memories. Subsequently, the data in the cache memory are overwritten; thus, write misses cause program execution delays.

Consider three store statements on a 2-dimensional array `a`: `a[i][j]=x`, `a[k][l]=y`, and `a[i][j+1]=z`. The first statement loads `a[i][j]` and the adjacent element `a[i][j+1]` into the cache. In this study, without loss of generality, we assume that the cache memory employs a direct mapping cache where each cache block corresponds to one address in the main memory and that the computer uses a write-back cache. If the second statement accesses data mapped to the same cache line, the cached elements are evicted. Consequently, the final statement must reload `a[i][j+1]` from main memory, resulting in a cache write miss. This miss can be avoided by executing accesses to adjacent elements consecutively. In this example, only access to the same array is considered. However, accessing different arrays requires accessing addresses in the main memory that are further apart. Therefore, to prevent write misses, it is crucial to move store statements in a manner that maintains consecutive accesses to the same array while maximizing the number of consecutive matching higher indexes.

In this study, we introduce a novel cache optimization algorithm called multi-dimensional global store statement aggregation (MDGSA) to reduce the number of write misses by consolidating store statements that reference the same arrays and leveraging the analysis of higher-dimensional continuity. MDGSA first analyzes whether any store statements are accessing other arrays between each store statement and subsequent store statements accessing the same array. As a result of this analysis, we identify regions where the same array could be accessed consecutively. Within these regions, MDGSA then analyzes the points where a higher number of consecutive matching of higher indexes occurs and delays the store statement at that point.

We implemented MDGSA in a compiler and compared the number of cache misses with those of existing code motion algorithms designed to reduce write misses. Consequently, our algorithm demonstrated a greater reduction in cache misses.

II. Preliminaries

We assume that all programs are represented by control flow graphs (CFGs). A CFG is represented as a quadruple consisting of a set of nodes **N**, a set of edges **E**, and start and end nodes **s** and **e**. Each node $n \in \mathbf{N}$ contains at most one statement. For each edge $(m, n) \in \mathbf{E}$, m is referred to as the predecessor of n, and n as the successor of m. As a node in a CFG can have multiple predecessors and successors, they are represented as sets. The sets of predecessors and successors of n are denoted as $pred(n)$ and $succ(n)$, respectively. Here, **s** and **e** are special nodes containing only empty statements.

The store statements moved by MDGSA are represented as assignment statements that access the arrays. That is, if a variable x is assigned to an array a with indexes i and j, it is expressed as $a[i][j] = x$. For simplicity, we make the following two assumptions. First, the left-hand side of the statement represents array access, and the right-hand side contains either a constant or a variable. In general, the right-hand side of a store statement may contain expressions such as function calls or array accesses. For these assignment statements, we introduce temporary variables to transform the right-hand side into a variable. For example, statements such as $a[i][j] = f()$ and $a[i][j] = b[i]$ are transformed into $t = f(); a[i][j] = t$ and $t = b[i]; a[i][j] = t$, respectively. Second, we represent all

store statements as array accesses, even though data can be stored in memory other than arrays, such as structures in C language.

While there are two write methods for cache memory, write-through and write-back, write-through involves writing to both the cache and main memories simultaneously, thus offering no performance improvement during write operations. As this study aims to reduce the number of accesses to the main memory during writes to the cache memory, it was designed assuming write-back cache memory. Finally, we do not prefetch the data into cache memory before executing the program.

Finally, we assume that the CFG is preprocessed to remove what are known as critical edges [1]. A critical edge (m, n) is defined as an edge where $\mid succ(m) \mid \geqq 2$ and $\mid pred(n) \mid \geqq 2$.

III. Related Work

This section reviews software-based cache optimization algorithms. Such algorithms are classified into approaches that use cache profile information [2] and those that do not. As MDGSA does not rely on profile information, we focus on the latter.

Existing compiler optimizations include global load instruction aggregation (GLIA) [3], [4] and GSA [5]. GLIA moves load statements to exploit spatial locality by maintaining consecutive accesses to the same array and increasing higher-dimensional index matching, similar to MDGSA. In contrast, GSA targets store statements and analyzes only array names without considering indexes. GLIA and MDGSA optimize load and store statements, respectively, and can therefore be applied together.

As MDGSA extends GSA by incorporating index analysis, its data-flow equations are built on GSA. Since GSA itself extends partial dead code elimination (PDE) [6], we first describe PDE, followed by GSA.

1) Partial Dead Code Elimination: PDE eliminates unused variable definitions by moving statements as close as possible to **e** along execution paths where the defined values are not used. First, candidate statements are identified using the predicate $Cand(n, v)$. Then, delayability analysis is performed using $Delayed$.

To analyze delayability, PDE first evaluates local predicates by inspecting each node. If a node uses the defined variable or redefines it, moving the statement beyond the node changes program semantics. These conditions are represented by $Used(n, v)$ and $Mod(n, v)$, respectively, and their disjunction defines $Block(n, v)$.

Using these local predicates, PDE evaluates global predicates across nodes. If no node between **e** and n satisfies $Used$, the variable v is regarded as dead, represented by $Dead(n, v)$. A node whose successor satisfies $Block(n, v)$ is represented by $Insert(n, v)$.

2) Global Store Statement Aggregation: GSA extends PDE for array store statements, represented as $ar = t$. Array movement requires considering both updates to the array and to index variables. To capture these effects, GSA introduces

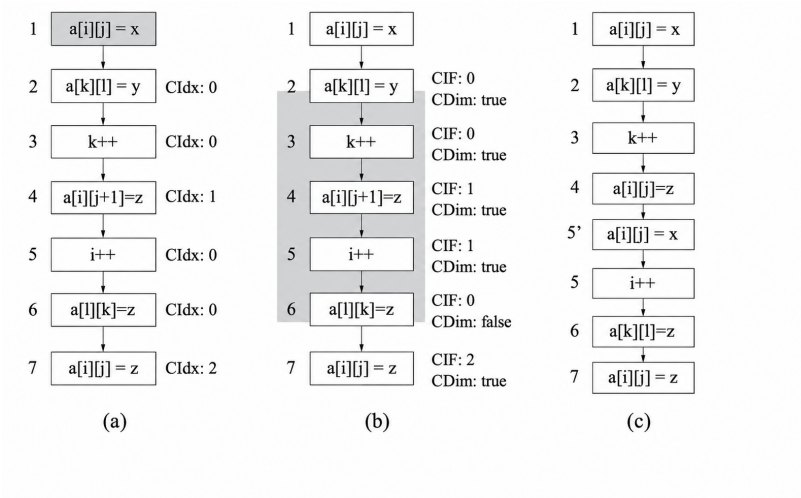

Fig. 1. (a) Local predicates for `a[i][j]=x` (b) Global predicates for `a[i][j]=x`. The gray area represents the range where the number of matched accesses to higher-order indexes is not reduced. (b) Result of MDGSA.

$Mod_{ar}(n, ar)$ and $Mod_{idx}(n, ar)$, representing modifications to the array and its index variables, respectively. Along with $Used(n, v)$, these predicates define $Block_s(n)$, which prohibits movements that change program semantics.

Unlike scalar variables, $ar = t$ can move across store statements updating different array elements. To distinguish such cases, GSA introduces $Store(n)$ and $isSame(n)$, indicating the existence of a store statement and whether it updates the same array element.

To aggregate consecutive accesses to the same array, GSA analyzes array names using $TopAddr$ and introduces $InCont$ and $Cont$, representing ranges where array accesses become discontinuous and continuous again. If these predicates hold at a node, moving ar before the node achieves consecutive access, represented by AAC.

IV. Array Index Analysis for Code Sinking

MDGSA extends GSA by analyzing the consistency of indexes; thus, the predicates defined by GSA are also utilized in MDGSA. We describe only newly defined predicates in MDGSA.

A. Local Predicates

MDGSA aggregates array accesses to maximize the continuity of higher-order indexes. For instance, as shown in Fig. 1 (a), the array access `a[i][j]` at Node 1 and the array access `a[i][j+1]` at Node 4 share a consistent higher-order index `i`. However, the array access `a[k][l]` at Node 2, which lies between these nodes, has different higher-order indexes from the previous two accesses. Therefore, by moving the array access `a[i][j]` from Node 1 to immediately after the access `a[k][l]` at Node 2, the likelihood of a write miss when accessing `a[i][j+1]` can be reduced.

To achieve this code motion, it is necessary to first count the number of consecutive matching higher-order indexes between

the array to be moved and the indexes of the other arrays. MDGSA formalizes this count with the predicate $CIdx$:

$$\begin{aligned} A &\overset{def}{\Leftrightarrow} a[i_1][i_2]...[i_p] \\ B &\overset{def}{\Leftrightarrow} b[j_1][j_2]...[j_q] \\ low &\overset{def}{\Leftrightarrow} min(p,q) \\ S &\overset{def}{\Leftrightarrow} \delta(TopAddr(A), TopAddr(B)) \\ CIdx(A,B) &\overset{def}{\Leftrightarrow} S \times \sum_{k=1}^{low} \prod_{l=1}^{k} \delta(Idx(A,l), Idx(B,l)) \quad (1) \end{aligned}$$

where $\delta(x,y)$ is a function that returns 1 if two arguments are the same; otherwise it returns 0. Function $Idx(A,l)$ returns the l-th index of the array given as the first argument. $CIdx(A,B)$, for two arrays given as arguments, first checks if they are the same array using S that is the result of the δ and the names of A and B. If they are in the same array, S is equal to 1; otherwise, it is 0. This result is useful for identifying different arrays: $CIdx$ always sets to zero for different arrays from ar. Subsequently, the consistency of the indexes is verified. The agreement of two indexes can be confirmed by combining the functions Idx and δ. As δ returns 1 if two arguments match, the consistency of all indexes from the highest to the k-th position can be determined by taking their production $\prod$. If there is a mismatch in the indexes along the way, multiplying by 0 results in a final production of 0. By performing this calculation from the highest to the lowest index and summing them, it is possible to compute whether the two arrays match the k-th index.

Fig. 1 (a) illustrates the computed values of $CIdx$ for each node when applying MDGSA to the array access `a[i][j]` in Node 1. Because there are no matching indexes with `a[k][l]` in Node 2, $CIdx$ is 0. However, with `a[i][j+1]` in Node 4, the higher-order index `i` matches, resulting in $CIdx$ being 1. The values for the other nodes are shown in the figure.

B. Global Predicates

MDGSA defines two new predicates CIF and $CDim$. The intuitive explanation for these predicates is that CIF propagates the number of matches calculated by $CIdx$ to the succeeding node each time a store statement accessing the array occurs. $CDim$ represents the range in which the propagated match count from the preceding node does not decrease. As MDGSA aims to move the store statement to a location where the match count is higher, it delays the store statement to the last node that satisfies $CDim$.

Fig. 1 (b) presents the values of the CIF and $CDim$ when MDGSA is applied to `a[i][j]` in Node 1. The detailed calculation is as follows. Initially, the count of matching indexes with the `a[k][l]` in Node 2 is 0, and this information propagates to subsequent nodes. Because there is no store statement in Node 3, the examination proceeds to Node 4. Node 4 has an array access `a[i][j+1]`; thus, the propagation of 0 stops at Node 3. This is achieved by setting $CIF(2,ar)$ and $CIF(3,ar)$ to 0. Similarly, as the count of matching indexes between the array accesses `a[i][j+1]` of Node 4 and `a[i][j]` is 1, this value is propagated to subsequent store statements until Node 5 is reached. In Node 6, although array access `a[l][k]` exists, the count of the matching indexes with `a[i][j]` is 0. Comparing the CIF in Node 6 with that in Node 5, the value decreases, indicating that the possible range for moving `a[i][j]` from Node 1 extends until just before Node 6. This comparison is performed using $CDim$; MDGSA sets $CDim(6,ar)$ as false. Utilizing this result, MDGSA sets $Delayed(6,ar)$ to false, determining the range where higher-order indexes continuously match until just before Node 6. Because the statement at Node 5 updates the value of variable `i`, the actual movable range extends only to immediately before Node 5. To enforce this movement restriction while preserving the index-continuity analysis, we extend the data-flow equation for $Delayed$ in GSA by incorporating $CDim$.

The data-flow equations for CIF and $CDim$ are defined as

$$mx \overset{def}{\Leftrightarrow} max(\{CIF(Str(m),ar) \mid m \in pred(n)\})$$

$$CIF(n,ar) \overset{def}{\Leftrightarrow} \begin{cases} 0 & \text{if } n = \mathbf{s} \\ CIdx(Str(n),ar) & \text{if } SameAddr(n) \\ mx & \text{otherwise} \end{cases} \quad (2)$$

$$CDim(n,ar) \overset{def}{\Leftrightarrow} \prod_{m \in pred(n)} CIF(n,ar) \geq CIF(m,ar) \quad (3)$$

MDGSA defines the predicate $Delayed$ as follows:

$$\begin{aligned} Delayed(n,ar) \overset{def}{\Leftrightarrow}\ & Cand(n,ar) \vee \\ & n \neq \mathbf{s} \wedge \neg Block_s(n) \vee \\ & AAC(n,ar) \wedge \sum_{s \in succ(n)} \neg AAC(s,ar) \vee \\ & \prod_{p \in pred(n)} CDim(p,ar) \wedge Delayed(p,ar) \quad (4) \end{aligned}$$

Fig. 1 (c) shows the results of the code motion for `a[i][j]=x`.

C. Application to the Overall Program

To move all store statements, MDGSA traverses the CFG in reverse topological order. When the visited node contains a store statement, MDGSA calculates the data-flow equations to facilitate code motion.

When MDGSA is applied the program shown in Fig. 1(a), the order of visiting nodes can be described as follows. As there are no branches in this graph, the nodes are visited individually from Node 7 to Node 1. Node 7 contains a store statement accessing `a[i][j]`; however, there are no subsequent nodes. Thus, MDGSA does not move the store statement. Next, Node 6 is visited. Although this node contains a store statement accessing `a[l][k]`, no other store statements access the same higher-level index `l` in subsequent nodes, and MDGSA does not perform specific actions regarding this store statement. Subsequently, MDGSA visits Nodes 5 and 4; however, there are no store statements that can be delayed. It then visits Nodes 3, 2, and 1, sequentially.

TABLE I
TNE NUMBER OF CACHE MISSES. BOLD LETTERS INDICATE THE BEST RESULTS.

Program	A. **PDE**	B. **GSA**	C. **MDGSA**	(A-C)/A	(B-C)/B
Matrix	103,638	101,913	**101,096**	2.45%	0.80%
Vector	108,517	106,786	**99,975**	7.87%	6.38%
Tensor	75,173	73,533	**72,840**	3.10%	0.94%

TABLE II
TNE NUMBER OF LLC STORE MISSES.

Program	A. **PDE**	B. **GSA**	C. **MDGSA**	(A-C)/A	(B-C)/B
Matrix	**82,281**	82,307	82,362	-0.10%	-0.07%
Vector	83,078	83,345	**82,317**	0.92%	1.23%
Tensor	59,015	**58,152**	58,169	1.43%	-0.03%

V. EXPERIMENTAL EVALUATION

A. Experimental Setting

To measure the effectiveness of store statement aggregation using MDGSA, we implemented MDGSA as a low-level intermediate representation converter of the COINS compiler[1].

The machine used for the evaluation was equipped with an Intel Core i7-11700 2.50GHz CPU and operated on an Ubuntu 64-bit operating system. This CPU utilized three levels of cache memory: L1D, L1I, L2, and L3 with sizes of 384 KiB, 256 KiB, 4 MiB, and 16 MiB, respectively.

We used three programs[2] named **Matrix**, **Vector**, and **Tensor** for evaluation. These programs compute the product of two matrices: the sum of two 3-dimensional vectors and the sum of two 3-dimensional tensors, respectively. The simple implementations of these programs have only one store statement for each array. Thus, we transformed the programs to ensure that four store statements appeared by applying loop unrolling to the iterations.

To evaluate the effectiveness of MDGSA, we implemented the following algorithms.

- **PDE** applies array reference extraction, eliminating critical edges, and PDE.
- **GSA** applies array reference extraction, eliminating critical edges, and GSA.
- **MDGSA** applies array reference extraction, eliminating critical edges, and MDGSA.

We collected the cache miss counts using performance analysis tools for Linux (perf command)[3]. We executed each program ten times and collected the number of cache misses. When discussing the results of the evaluation in this study, we present the average values.

B. Results

Tables I and II present the total cache and store miss counts in the L3 cache memory, respectively. First, focusing on the total cache miss counts, MDGSA exhibited fewer misses in all programs than that of PDE and GSA. Particularly for **Vector**, MDGSA reduced miss counts by approximately 6∼8%. Examining the store miss counts in the L3 cache memory, MDGSA achieved better results than the PDE and GSA algorithms by approximately 1% in **Vector** and by approximately 1% over PDE in **Tensor**. These results suggest that code aggregation for multidimensional arrays using MDGSA reduces the number of cache misses compared to PDE and GSA, particularly in terms of reducing misses in L1 and L2 cache memories.

[1]https://sourceforge.net/projects/coins-project/

[2]The evaluated programs are available at: https://bit.ly/3T74tlh

[3]https://manpages.ubuntu.com/manpages/kinetic/man1/perf.1.html

VI. CONCLUSIONS & FUTURE WORK

We proposed MDGSA, a cache optimization algorithm that reduces write misses by moving store statements to enable consecutive accesses with maximum higher-dimensional index matching. Unlike existing methods, MDGSA analyzes not only array names but also index matching from higher dimensions. To realize this movement, we define data-flow equations that identify locations maximizing index matching within consecutive accesses. We evaluated MDGSA, PDE, and GSA on three programs using multi-dimensional arrays by measuring cache misses. The results show that MDGSA reduces cache misses compared with existing algorithms across all cache levels.

A potential future research direction involves simultaneously moving the store and load statements. While GLIA and MDGSA allow for separate movements of these statements, each is specialized for load or store statements, respectively. Consequently, no consolidated analysis of access destinations for store or load statements has been conducted. In addition, we will propose a method that uses the profile information of actual write misses measured using hardware. Some code optimizations, particularly those used in just-in-time compilers, utilize information obtained by executing programs. Such information can be used to further improve execution efficiency, particularly for code motion methods specific to array references, which generate many cache misses.

REFERENCES

[1] J. Knoop, O. Rüthing, and B. Steffen, "Lazy code motion," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 1992, pp. 224–234.

[2] J. Lifflander and S. Krishnamoorthy, "Cache locality optimization for recursive programs," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2017.

[3] Y. Sumikawa and M. Takimoto, "Global load instruction aggregation based on code motion," in *2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming*, 2012, pp. 149–156.

[4] ——, "Global load instruction aggregation based on dimensions of arrays," *Computers & Electrical Engineering*, vol. 50, pp. 180–199, 2016.

[5] T. Sano and Y. Sumikawa, "Global store statement aggregation," in *2023 IEEE 14th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, 2023, pp. 1–6.

[6] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," *SIGPLAN Not.*, vol. 29, no. 6, pp. 147–158, jun 1994.