Global Store Statement Aggregation

Tomohiro Sano
Takushoku University
Tokyo, Japan
sumikawa.lab@gmail.com

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

***Abstract*—The memory hierarchy comprises main memory, which can store a large amount of data, and cache memory, which can access data at high speeds. By accessing data stored in the same array continuously, we can use cache memory without performing frequent data replacement from the main memory, which is crucial for high-speed execution of programs. In this study, we propose a novel code motion algorithm, named global store statement aggregation (GSA), to improve program execution by efficiently utilizing the memory hierarchy. To achieve this, GSA moves each store statement referring to an array immediately before the following store statements accessing the same array. We implemented GSA in a real compiler and evaluated it for sorting programs. The experimental results indicate that our algorithm effectively reduces the number of cache misses in comparison to a previous code motion algorithm.**

***Index Terms*—Code optimization, cache memory, code motion, store instruction**

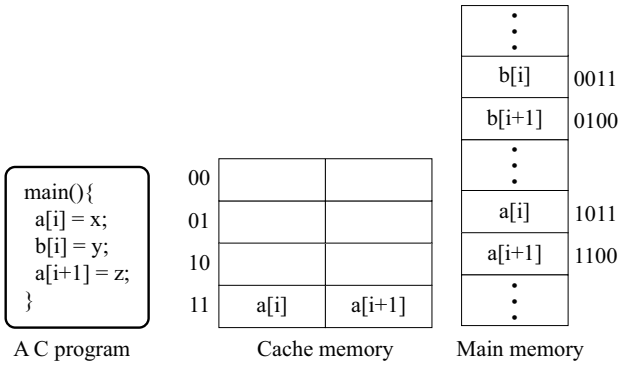

Fig. 1. Example of the kind of cache miss on which our algorithm is focused.

I. Introduction

Most processors use cache memory and main memory. If a processor stores the data to memory, it first checks cache memory if there is an old data from the same memory address. If the data exists, it is called *write hit*; otherwise, it is called *write miss*. There are two types of cache write policies: write-through and write-back. In either method, if a write miss occurs, it is necessary to perform cache line replacement to maintain data consistency. This replacement requires accessing the main memory; thus, the execution time of the program becomes slower. This indicates that continuous accesses to memory addresses close to each other, such as accessing to the same array, are important to keep execution time fast.

Fig. 1 shows a C program describing how the cache line replacement is performed. We here have following assumptions that 1) the cache memory is designed as direct-mapped cache although we can say the same logic to n-way associative, 2) the cache write policy is write-back, and 3) any prefetch is not executed. Looking at the C program, the first executed statement is to store `x` to `a[i]`. As there is no `a[i]` data on the cache memory at first, the processor copies the data with the other data around `a[i]` from the main memory to the cache memory. It then executes the store statement `b[i]=y`. In the case where accessing `b[i]` requires the same cache line as `a[i]`, this execution causes a write miss; the processor replace the `a[i]` and `a[i+1]` with `b[i]` and `b[i+1]`. After this replacement, the last statement `a[i+1]=z` is executed. This execution requires reloading `a[i]` and `a[i+1]` from the main memory because the previous store statement `b[i]=y` excluded the `a[i+1]` from the cache memory. If this program switches the execution order of `a[i]` and `b[i]`, it is possible to prevent causing the write miss.

In this example, the replacement process was relatively simple; however, in actual usage, data is overwritten only in cache memory in case of a write hit. If a write miss occurs when there is an inconsistency between the cache memory and the main memory, additional processes are performed to ensure that the consistency between the two types of memories is not compromised.

In this study, we propose a novel cache optimization algorithm to reduce the number of write misses by aggregating store statements referring to the same arrays. We name the proposed algorithm global store statement aggregation (GSA). GSA delays executions of store statements without losing continuous accessing to the same array. GSA analyzes the delayability of each store statement one by one from the end of the program to the start. We design this code motion as an extension of partial dead code elimination (PDE) that sinks assignment to eliminate dead codes [1].

Looking back to the example shown in Fig. 1, GSA first analyzes the delayability of `a[i+1]=z`. As this statement is the last, GSA does not delay it. GSA next analyzes whether `b[i]=y` is delayable. Although this statement is delayable, GSA does nothing to the statement because there is no following store statement referring to the same array. Finally, GSA analyzes `a[i]=x`'s delayability. Moving this statement beyond statement `b[i]=y` would result in store statement `a[i+1]=z`, which follows `b[i]=y`, accessing the same array

continuously.

To confirm the effectiveness of GSA, we implemented GSA on a compiler infrastructure called COINS. We evaluated how much continuous access ordering of store statements by GSA reduced cache misses and improved execution times of objective codes. This evaluation confirmed that GSA reduced the number of cache misses and reduced execution time compared with PDE.

II. Related Work

The effectiveness of cache memory is widely recognized, and therefore, research is being conducted on architectural designs that utilize cache memory, e.g., [2], and on improving the efficiency of cache memory utilization. Focusing on the latter research to improve cache memory utilization efficiency, several methods have already been proposed as software and hardware approaches. As GSA is a software approach-based cache optimization, we here focus on previous software based methods; see survey papers, e.g., [3] to see the hardware based approaches.

Given the prevalence of multi-core processors and GPUs in modern computing systems, various methods have been proposed to achieve efficient cache reuse, even in complex hardware configurations, by leveraging information from actual program execution. Lifflander and Krishnamoorthy proposed an optimization algorithm for recursive programs [4]. This algorithm collects interference and dependencies by tracking executions. As the information exploits data reuse opportunities, the algorithm then applies work-stealing scheduler. Tripathy *et al.* proposed PAVER that is a priority-aware vertex scheduler [5]. PAVER improves cache locality among thread blocks on GPU by using execution profiling information. The information is used for creating a graph whose nodes and edges are corresponding to thread blocks and data sharing statistics, respectively. PAVER performs graph partitioning for maximizing cache sharing within processors while maintaining load balance between processors. The use of execution information, as in these techniques, is effective for programs that can be executed repeatedly. On the other hand, as GSA does not require execution information, it can work effectively in environments where such information is not available. In addition, these profile-guided methods and GSA are usable at once because GSA improves spatial locality before applying the profile-guided ones; it is thought that the effectiveness of other methods can be further enhanced.

Looking at compiler optimization methods, a code motion-based cache optimization technique, named global load instruction aggregation (GLIA), is proposed [6, 7]. GLIA is a method making continuous accesses to the same arrays as well as GSA; however, its target is different from GSA. GLIA moves load statements accessing array references respecting their access order to reduce the number of read miss by aggregating array references. As GSA moves only store statements, the combination of GSA and GLIA allows for many array references accessed continuously.

III. Background

This section first presents definitions GSA assumes. It then describes PDE to present how PDE sinks each statement.

A. Preliminaries

Program representation. GSA assumes that all programs are represented as intermediate representation. To ease presentation, we represent all load and store statements as accessing arrays, e.g., $a[i]$ with address a and index i. Although the statements may access structures in C language, GSA improves their localities as well as array references. We represent load statement as $x = a[i]$; that is, the right-hand side of statement loads a data from array $a[i]$ into a temporal variable x. On the other hand, we represent store statement as $a[i] = x$; that is, the right-hand side of statement stores a data from a temporal variable x into array $a[i]$.

Array reference extraction. GSA also assumes that the right-hand side of store statement is a temporal variable. If a function return value is stored into an array such as $a[i] = f()$, GSA splits it into $t = f()$ and $a[i] = t$. As another example, if a statement includes array references in both right- and left-hand sides of an assignment such as $a[i] = a[j]$, GSA splits it into $t = a[j]$ and $a[i] = t$.

Control flow graph. Before applying GSA, we assume that each program has been represented as a control flow graph (CFG). A CFG is a quadruple (**N**, **E**, **s**, **e**). **N** is a set of nodes with a single statement. **E** is a set of directed edges connected to the nodes. Each edge is represented as $(m, n) \in \mathbf{E} \subset \mathbf{N} \times \mathbf{N}$, where m and n are called a predecessor of n and a successor of m, respectively. In general, there are several predecessors and successors for a node, because of the nondeterministic branching structure of a CFG. The sets of predecessors and successors of node n are denoted by node sets $pred(n)$ and $succ(n)$, respectively. **s** and **e** are a start and an end nodes with empty statements, respectively.

Similar to other code motion algorithms [8], CFG excludes all *critical edge* that leads from a node with more than one successor to a node with more than one predecessor by inserting a synthesized node to each critical edge, because critical edges can block code motion.

B. Partial Dead Code Elimination

PDE is an extension of dead code elimination (DCE) [9] that eliminated only totally dead assignments to be able to eliminate partially dead assignments. If a variable is defined in an assignment, but there is no usage of the variable on all execution paths from the definition point to **e**, then the variable and the assignment are called totally dead or simply dead. If a variable is used on several execution paths, but not all, from the definition point to **e**, the variable and the assignment are called partially dead.

Figs. 2 and 3 show how PDE eliminates partially dead codes by delaying assignments. Looking at the Fig. 2, Node 1 has an assignment `y=a+b`. This variable `y` is used on Node 4 if the control flow includes Node 3. By contrast, it is not used on a path through Node 2 as Node 2 has another assignment

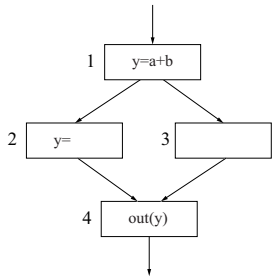

Fig. 2. An original program

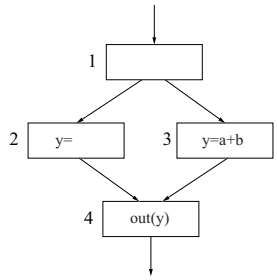

Fig. 3. Result of PDE

defining y; thus, the variable y is partially dead. PDE sinks y=a+b from Node 1 to immediately before Nodes 2 and 3, and then eliminates the sunk assignment on Node 2. As a result, PDE obtains the CFG shown in Fig. 3. Thus, PDE eliminates partially dead by performing two steps: code sinking and DCE.

We now present formal definitions to perform PDE. In the definition, we regard $v = t$ as a partially dead assignment eliminated by PDE. PDE defines three predicates: $Dead(n, v)$, $Delayed(n, v)$, and $Insert(n, v)$. $Dead(n, v)$ represents that v is considered as totally dead. $Delayed(n, v)$ means that v is sinkable at node n. In the following, we use sink and delay interchangeably; we use sink to describe the intuitive behavior of the algorithm and delay to describe the definition of the data-flow equation. $Insert(n, v)$ indicates that PDE inserts v at n. PDE iteratively performs DCE using $Dead(n, v)$ and assignment sinking using $Delayed(n, v)$ and $Insert(n, v)$ until the program becomes invariant.

These predicates are calculated by using two local predicates $Used(n, v)$ and $Mod(n, v)$. $Used(n, v)$ indicates that node n includes an statement containing v without updating value of v. $Mod(n, v)$ indicates that node n includes an statement updating a value of v. These predicates mean that the sinking of v beyond that nodes may change the meaning of the program; the sinking must be blocked. To preserve program semantic, PDE forbids code sinking beyond statements that uses v or modifies x or t. The predicate $Block(n, v)$ represents this forbidding. The formal definition of $Block(n, v)$ is given as follows:

$$Block(n, v) \overset{def}{\Leftrightarrow} Used(n, v) \vee Mod(n, v)$$

Looking at Fig. 2, Node 2 has a statement y= that modifies the variable v that is a left-hand side of the sinking statement y=a+b. PDE sinks the statement before the Node 2, and then eliminates the statement because it becomes totally dead. The deadness is calculated as follows: 1) it first checks if the result of sinking before the Node 2 has not any statement using y, and 2) its successor 2 modifies the y; thus, $Dead(2, y)$ becomes true. These results indicate that the immediately following statement makes the sunk assignment dead; the data-flow equation about $Dead(n, v)$ captures this. The predicate $Delayed$ represents the delayability. It first analyzes that there is a target statement represented by $Cand(n, v)$. It then sinks as close as possible to the **e** by analyzing whether there is no blocking statements or whether successors also permits the sinking. As a result of the delayability analysis, PDE inserts the statement to the closest nodes to **e** represented by $Insert$.

The data-flow equations of three predicates $Dead(n, v)$, $Delayed(n, v)$, and $Insert(n, v)$ are defined as follows:

$$\begin{aligned}
Dead(n, v) &\overset{def}{\Leftrightarrow} \neg Used(n, v) \wedge \\
&\quad (Mod(n, v) \vee \prod_{s \in succ(n)} Dead(s, v)) \\
Delayed(n, v) &\overset{def}{\Leftrightarrow} Cand(n, v) \vee \\
&\quad (n \neq \mathbf{s}) \wedge \neg Block(n, v) \wedge \prod_{p \in pred(n)} Delayed(p, v) \\
Insert(n, v) &\overset{def}{\Leftrightarrow} Delayed(n, v) \wedge \\
&\quad (Block(n, v) \vee \sum_{s \in succ(n)} \neg Delayed(s, v))
\end{aligned}$$

IV. Sinking Array References

This section presents the detail algorithm of GSA extended by PDE. The intuitive idea is to extend $Delayed$ to check the order of arrays referenced from the store statement and to make the access order continuous. To achieve the idea, we introduce local predicates for checking the array addresses and global predicates for preserving the order of array references. In this section, we assume that GSA is analyzing predicates for $ar = t$ on node n.

A. Local Predicates

To check array addresses and delayability, GSA introduces five local predicates $Store(n)$, $isSame(n)$, $Mod_{idx}(n, ar)$, $Mod_{ar}(n, ar)$, and $Block_s(n)$.

Fig. 4 shows how GSA analyzes local predicates for sinking the store statement a[i]=x on Node 1. This store statement should be sunk as the following store statement b[i]=y breaks the accessing to array a consistency. To invoke code sinking, the GSA initially examines the local predicates in order to determine the node at which a store statement exists, the array to which the store statement refers, and the potential presence of any statements that may block sinking. Looking at Node 2, there is a store statement b[i]=y and its accessing array is different from a[i]=x. For representing these analyzing results, GSA sets $Store(2)$ as true and $Mod_{ar}(2, \mathtt{a})$ as false. The Mod_{ar} indicates that any of the elements of the analyzing array ar is modified; the predicate should be false as the store statement b[i] does not change the value of array a. Next, GSA analyzes Node 3. This node updates the value of the variable used in a[i]. To represent this update, GSA sets $Mod_{idx}(3, \mathtt{a})$ as true because this statement defines a value of i used as the index of a[i]. Finally, Node 4 includes a store statement accessing to a. GSA sets $Store(4)$ and $Mod_{ar}(4, \mathtt{a})$ as true.

The predicate $Store(n)$ represents that n includes a store statement. The predicate $isSame(n)$ analyzes that the state-

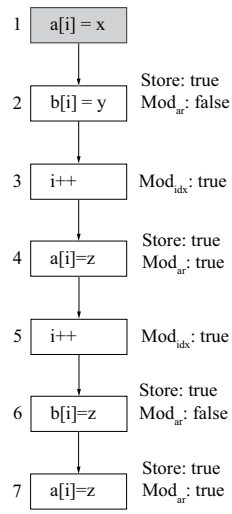

Fig. 4. Local predicates for `a[i]=x`

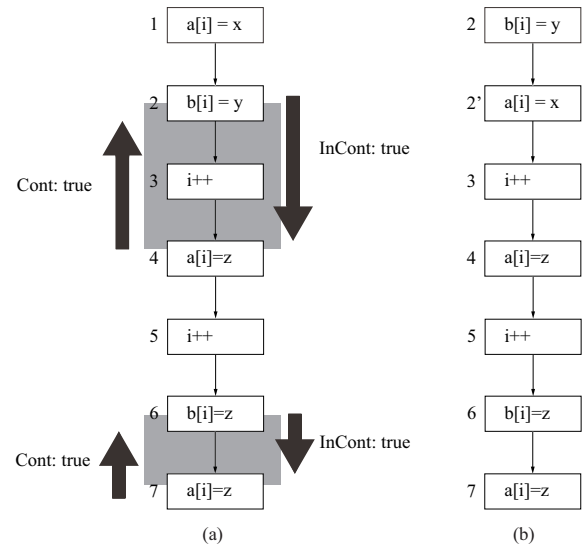

Fig. 5. (a) Global predicates for `a[i]=x`. The gray area represents the range where access to the array `a` becomes continuous when this store statement is sunk. (b) Result of GSA.

ment of n is a store statement and its left-hand side is lexical equal to ar. This is formally defined as follows:

$$isSame(n) \overset{def}{\Leftrightarrow} Store(n) \wedge lhs(n) = ar$$

where $lhs(n)$ is a function returning the left-hand side of assignment statement. If the statement of n is not assignment statement, the function returns $\perp$. Thus, this predicate first checks whether n has a store statement. It then checks if the left-hand side of the statement equals to ar.

$Mod_{ar}(n, ar)$ also analyzes that the statement of n is a store statement but it analyzes only the equality of the array address a. The formal definition is given as follows:

$$\begin{aligned} Addr_l(n) &\overset{def}{\Leftrightarrow} TopAddr(lhs(n)) \\ Addr_{ar} &\overset{def}{\Leftrightarrow} TopAddr(ar) \\ Mod_{ar}(n, ar) &\overset{def}{\Leftrightarrow} Store(n) \wedge Addr_l(n) = Addr_{ar} \end{aligned}$$

where $TopAddr(ar)$ returns the start address of ar if ar is an array; otherwise, it returns $\perp$.

When sinking a store statement, we need consider two types of changes: changing the data in the array and changing the value of a variable used as an array index. To capture the latter case, we define Mod_{idx} as follows:

$$Mod_{idx}(n, ar) \overset{def}{\Leftrightarrow} Def(n, ar) \in Var(ar)$$

where $Var(ar)$ is a function extracting variables used in ar. For example, when $Var(a[i])$ is performed, the function returns i.

$Block_s(n)$ represents that n includes blocking statement for ar and variables used in ar. This predicate is formally defined as follows:

$$\begin{aligned} Mod_{gsa}(n, ar) &\overset{def}{\Leftrightarrow} Mod_{ar}(n, ar) \vee Mod_{idx}(n, ar) \\ Block_s(n, ar) &\overset{def}{\Leftrightarrow} Used(n, ar) \vee Mod_{gsa}(n, ar) \end{aligned}$$

B. Global Predicates

GSA introduces a novel global predicates $Cont$ and $InCont$ and extends $Delayed$ of PDE to use them. The intuitive idea of $Cont$ is that the first following store statement refers to the same array as ar. On the other hand, $InCont$ represents that the immediately preceding store statement refers to the different array from ar. By using these predicates, it is possible to analyze the range of continuous accesses to the same array by sinking of GSA.

Fig. 5 (a) shows the results of global predicate analyses of GSA for the store statement on Node 1. GSA first analyzes the $Cont$ predicate to identify the start points of ranges that can continuously refer to ar between **e** and the $ar = t$. It then identifies the end points of the ranges represented by the $InCont$ predicate. In this example, Node 4 includes an array access to `a` and the continuous access to the array continues since Node 2. After analyzing the array access flows, GSA finds three points on Node 3: 1) the access order is broken in the forward direction, 2) the access order is preserved in the backward direction, and 3) the code sinking beyond Node 3 is prohibited. These results indicates that sinking `a[i]=x` to immediately before Node 3 makes array accessing to `a` continuous without changing the semantics of the program as shown in Fig. 5(b).

GSA modifies $Delayed$ to check whether the *array access continuous* condition, represented by AAC, is satisfied. As the two predicates $Cont$ and $InCont$ identify the ranges that can continuously refer to ar, sinking $ar = t$ into the ranges satisfies the condition; thus, this condition is formally defined

as follows:

$$Cont(n, ar) \overset{def}{\Leftrightarrow} Mod_{ar}(n, ar) \vee \neg Mod_{ar}(n, ar) \wedge \sum_{s \in succ(n)} Cont(s, ar)$$

$$InCont(n, ar) \overset{def}{\Leftrightarrow} \begin{cases} \neg Mod_{ar}(n, ar) & \text{if } Store(n) \\ \sum_{p \in pred(n)} InCont(p, ar) & \text{otherwise} \end{cases}$$

$$AAC(n, ar) \overset{def}{\Leftrightarrow} InCont(n, ar) \wedge Cont(n, ar)$$

GSA uses the AAC as follows:

$$Delayed(n, ar) \overset{def}{\Leftrightarrow} \begin{cases} \text{true} & \text{if } Cand(n, ar) \\ \text{false} & \text{else if } n = \mathbf{s} \\ \text{false} & \text{else if } Block_s(n, ar) \\ \sum_{s \in succ(n)} \neg AAC(s, ar) & \text{else if } AAC(n, ar) \\ \prod_{p \in pred(n)} Delayed(p, ar) & \text{otherwise} \end{cases}$$

C. Application to the Overall Program

GSA performs the sinking analysis for each store statement. GSA finds the statement one by one with reverse topological sort order traversing on CFG. This analysis style is called demand-driven analysis. Traditional PDEs are designed as an exhaustive analysis that determines the deadness/faintness of all expressions using bit vectors and checking lexical equality. In general, application of code sinking exposes new dead/faint codes, known as the *second order-effect*. Thus, to discover many dead/faint codes, many PDEs must be applied; however, reflecting the second order-effect requires more analysis time [1]. By contrast, demand-driven analysis is proposed to reflect many of the second order-effects in one application [10].

Figs. 6 $\sim$ 8 show the order in which GSA sinks the store statements. Looking at Fig. 6, GSA first checks whether there is a store statement on Node 5. This node has a store statement; however, it does nothing as this node does not have any successors. It next checks the existence of a store statement on Node 4. As there is no any statements on this node, it then checks Node 3. Node 3 does not include any store statement, GSA skips code sinking on the node. GSA finds a store statement `b[i]=y` on Node 2; thus, GSA analyzes the local and global predicates, and then sinks it to immediately before Node 5 in order to make accessing to `b` continuous as shown in Fig. 7. Finally, GSA finds a store statement `a[i]=x` on Node 1. As this store statement is partial dead, GSA sinks this statement to immediately before Node 3. As results of these code sinking, GSA obtains the program shown in Fig. 8.

V. Experimental Evaluation

A. Experimental Setting

We implemented GSA as a low-level intermediate representation converter in a COINS compiler[1].

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

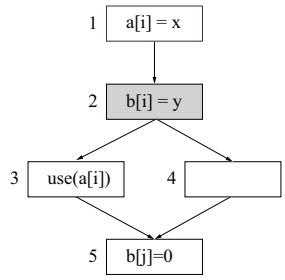

Fig. 6. An original program

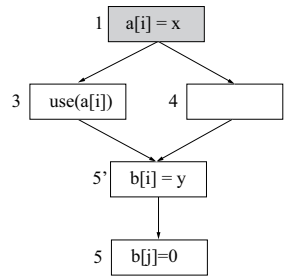

Fig. 7. Sinking `b[i]`

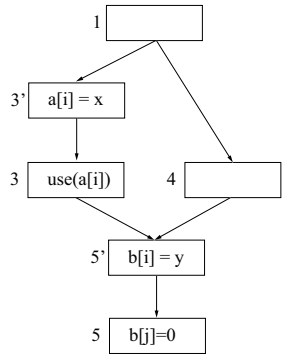

Fig. 8. Sinking `a[i]`

TABLE I
TNE NUMBER OF CACHE MISSES.

Program	A. **PDE**	B. **GSA**	(A-B)/A
Count	25,480	25,266	0.84%
Radix	6,662	6,575	1.31%
Array	18,362	17,343	5.55%

We conducted the experiments on a machine equipped with a Intel Corei7-11700 2.50GHz CPU and an Ubuntu 64bit operating system. This CPU has three leveled cache memories, L1d and Li1, L2, and L3 cache memories. Their sizes are: 384 KiB, 256 KiB, 4 MiB, and 16 MiB, respectively.

We use three programs[2] named **distcountsort** (**Count**), **radixsort** (**Radix**), and **arrays** (**Array**) for the evalution. The **Count** and **Radix** are implementations of counting and radix sorts, respectively. The **Array** is the program shown in Fig. 1. We have made modifications to the program of the figure to enable it to operate within a loop structure.

To evaluate the effectiveness of GSA, we used **PDE** that applies array reference extraction, eliminating critical edges, and PDE as a baseline.

B. Results

Tables I and II show the number of all cache misses and last level cache (LLC) for the two algorithms, respectively. We obtained these cache miss numbers by Performance analysis

[2]The evaluated programs are available at: https://drive.google.com/drive/folders/12QmcA-gcyeak6TXGaetEMwfpCXf0jf6O?usp=sharing

TABLE II
TNE NUMBER OF LLC STORE MISSES.

Program	A. **PDE**	B. **GSA**	(A-B)/A
Count	17,918	17,889	0.16%
Radix	1,200	1,171	2.42%
Array	11,661	11,450	1.81%

TABLE III
EXECUTION TIMES. THE UNITS IS SECOND.

Program	A. **PDE**	B. **GSA**	(B-A)/A
Count	1564.8	1,342	14.24%
Radix	566.6	562.5	0.72%
Array	551.5	531.5	3.63%

tools for Linux (perf command)[3]. In both types of cache misses, we can see that GSA obtained lower numbers of cache misses for all three programs than PDE. In particular, for the **Array** program, all cache misses were reduced by approximately 5%. As the miss rate reduction for L3 cache memory was approximately 1.8%, it can be seen that GSA especially reduced the misses for L1 and L2 cache memories. Focusing on misses in L3 cache memory, GSA reduced the miss rate by approximately 2.4% for the **Radix** program.

Next, we evaluated the impact of reducing cache misses for the execution time of the objective code. Table III shows the times. In all programs, GSA achieved better results than PDE. Particularly, **Count** showed the most significant improvement in execution time, achieving a reduction of approximately 14% compared to PDE. On the other hand, **Radix**, which was able to reduce LLC misses the most, only achieved a reduction of about 0.7% in execution time. To understand the reason better, we checked these programs in detail.

We found that GSA moved one store statement for each of the **Radix** and **Count**. Looking at **Radix**, there are three store statements as follows: `tmp[box[kakunou[0]]]=data[i]; box[kakunou[1]]--; tmp[box[kakunou[1]]] = data[i-1];` These store statements alternately access two arrays, `tmp` and `box`; thus, sinking the first store statement accessing to `tmp` beyond the `box` makes continuous accesses to `tmp`. The **Count** program has the following statements. `res[box[data[i]]] = data[i]; box[data[i-1]]--; res[box[data[i-1]]] = data[i-1];` GSA moved the first statement of `res` beyond the `box`.

When comparing the two programs, the array reference used in the load statement in **Radix** had a constant index, while in **Count**, it was a variable index. We believe that GSA improved the temporal locality of memory accesses by aggregating stores. However, the impact on the execution time of the objective code was influenced by the variation in the spatial locality caused by differences in memory addresses accessed each time the loop is executed.

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

VI. CONCLUSIONS & FUTURE WORK

In this paper, we proposed a novel code motion based cache optimization algorithm, named global store statement aggregation (GSA). GSA aims at reducing write misses by making store statements accessing the same array continuously. We have evaluated the effectiveness of GSA by measuring the number of cache misses and the execution times of objective codes. We confirmed that GSA obtained the best result compared to previous code motion algorithms.

In future works, we will examine *moving all store and load statements at the same time.* GSA and GLIA have been proposed as methods to do these things, respectively. We would like to investigate their combination in depth.

REFERENCES

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

[2] L. Zhou, B. Lu, S. Zhang, and L. Qi, "Data cache optimization model based on hbase and redis," in *Proceedings of the 3rd International Conference on Data Science and Information Technology*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 31–35.

[3] W. Zang and A. Gordon-Ross, "A survey on cache tuning from a power/energy perspective," *ACM Comput. Surv.*, vol. 45, no. 3, jul 2013.

[4] 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.

[5] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, and D. Wong, "Paver: Locality graph-based thread block scheduling for gpus," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, jun 2021.

[6] 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.

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

[8] 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.

[9] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[10] Y. Sumikawa and M. Takimoto, "Effective demand-driven partial redundancy elimination," *Inform Process Soc Jpn Trans Programm*, vol. 6, no. 2, pp. 33–44, aug 2013.