Global Load Instruction Aggregation Based on Dimensions of Arrays

Yasunobu Sumikawa[a,*], Munehiro Takimoto[a],

[a] *2641, Yamazaki, Noda, Chiba, Japan, 278-8510, Tokyo University of Science*

Abstract

Most modern processors have some cache memories that are much faster than main memory. These cache memories function well if temporal or spatial localities in programs are enhanced; therefore, the continuous accesses to the same array that improves the localities can enhance effective utilization of the cache memories. In addition, because a multidimensional array can be regarded as an array of lower dimensional arrays, the continuous accesses to array references with the most similar indexes can enhance the utilization of them further. We propose a new code motion algorithm called MDGLIA that improves utilization of cache memory. MDGLIA moves each array reference immediately after the preceding references accessing the same array with the most similar indexes, and then delays it as late as possible without changing the access order. These two-step code motions contribute to not only improvement of the cache efficiency in the overall program but also suppression of register pressure. We implemented MDGLIA in a real compiler and evaluated it for matrix multiplication and SPEC benchmarks. The experimental results show that our algorithm reduces the number of cache misses in most cases.

Keywords: partial redundancy elimination, cache optimization, register pressure, and data-flow analysis

1. Introduction

Most modern processors have some cache memories that are much faster than main memory. Whenever a processor needs the data at address x in main memory, the cache memory is checked first to ascertain whether a copy of the data is stored in cache memory. At this time, it is called *cache hit* if the data at x is found in the cache memory; otherwise, it is called *cache miss*. In the case of a cache hit, because the data is obtained without any memory access, the program is executed without stalling. Conversely, when a cache miss occurs, the processor fetches the data around x in the main memory, and then places it into cache memory for subsequent cache hits. In this case, the reference to x not only causes significant delay because of the fetching and placing of the data around x into cache memory but also removes the old data in the same cache line. This means that continuous access to addresses that are at a distance from each other in main memory may result in cache misses, which can reduce the execution efficiency of the program.

Let us look at how accessed data is copied into cache memory. Consider the C program outlined in Figure 1. In this paper, for ease of explanation, we assume that the cache memory is directly mapped without loss of generality. That is, when the data is transferred from main memory to cache memory, the candidate location in which it is placed is determined by the memory address modulo and the number of lines in the cache mem-

*Corresponding author

Email addresses: yas@cs.is.noda.tus.ac.jp (Yasunobu Sumikawa), mune@cs.is.noda.tus.ac.jp (Munehiro Takimoto)

URL: http://www.cs.is.noda.tus.ac.jp/~yas (Yasunobu Sumikawa)

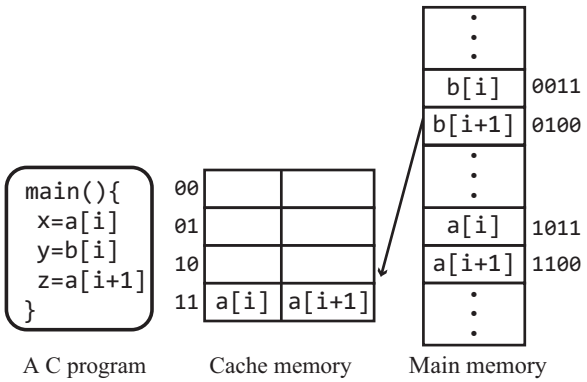

Figure 1: Example of the kind of cache miss on which our algorithm is focused.

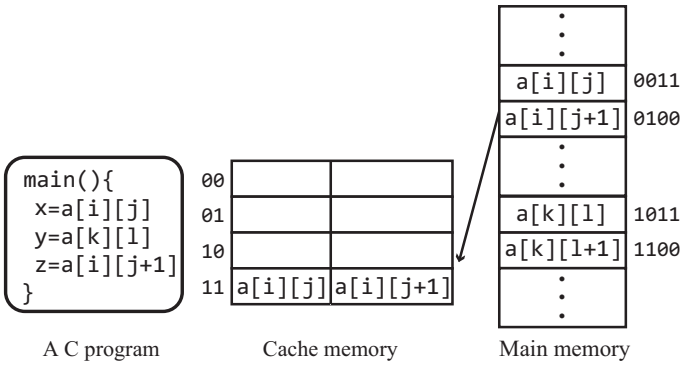

Figure 2: Example of the kind of cache miss on which our algorithm is focused.

ory. Following execution of array reference `a[i]`, the data at `a[i]` and `a[i+1]` will be copied on the cache line with index `11`. Subsequently, because execution of array reference `b[i]` results in a cache miss, the data at `b[i]` and `b[i+1]` will also be copied into cache memory. In this case, the data is copied on index `11` because the address at `b[i]` is `0011`; therefore, the data at `a[i]` and `a[i+1]` will be removed from the cache memory, which may result in a cache miss for subsequent access to `a[i+1]`.

As shown in the example, once the data at a specific array index is loaded from main memory, it is placed in the cache memory along with other data belonging to the same array. That is, continuously accessing the same array can get cache hits. Continuous access to the same array can be promoted by moving references to an array around other references to the same array. In Figure 1, moving `a[i+1]` immediately before `b[i]` can prevent a cache miss because `a[i+1]` could then be executed immediately after `a[i]`.

Furthermore, considering that a multidimensional array represents an array of lower dimensional arrays, preferentially aggregating references with the same indexes more in higher dimensions may further reduce cache misses. Consider the array reference `a[i][j+1]` in Figure 2. The referenced data is copied to cache memory following the execution of array reference `a[i][j]`. However, as was the case with the program in Figure 1, execution of `a[k][l]` may cause the data to be expunged, resulting in a cache miss for `a[i][j+1]`. However, this cache miss can be prevented by moving the array reference immediately before `a[k][l]`.

We present a new cache optimization algorithm that continuously aggregates array references with the same indexes to higher dimensions. The proposed algorithm, called the multidimensional global load instruction aggregation (MDGLIA), is based on *partial redundancy elimination* (PRE) [3, 15, 18] that makes partially redundant expressions fully redundant by inserting some expressions, and then removing the redundant expressions. MDGLIA extends PRE to aggregate array references, without sacrificing the effects of gained by removing redundant expressions. MDGLIA determines the number of indexes of each array reference *ar* preceding a moved candidate that are the same array as *ar*, and then moves *ar* to the program points closest to the references with the same indexes most in higher dimensions.

Consider the array references in Figure 3(a). MDGLIA is applied to each array reference traversing the control flow graph (CFG) in the *topological sort order*. First, MDGLIA moves array reference `a[k][l]` immediately before array reference `b[i]` because the execution of `b[i]` between `a[i][j]` and `a[k][l]` may cause `a[k][l]` to be removed from the cache memory if the data placed in the cache memory for `b[i]` share some cache lines for `a[k][l]`. Here, MDGLIA makes a new Node 6' for the moved array reference. Next,

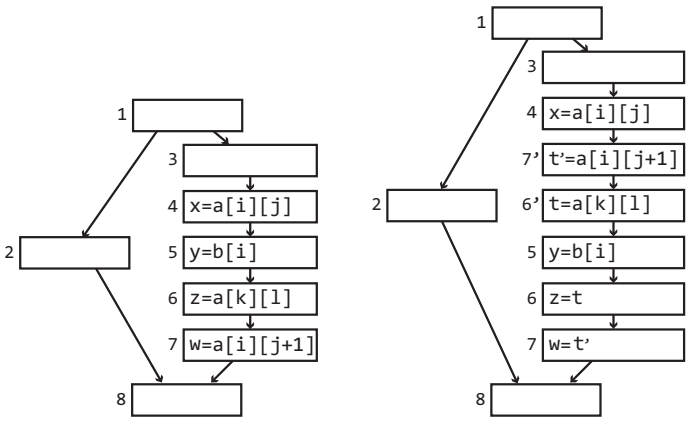

Figure 3: Effectiveness of MDGLIA: (a) Original code. (b) Result of applying MDGLIA.

consider `a[i][j+1]` at Node 7. Although sub-array `a[i]` is accessed at Nodes 4 and 7, another array, `b`, and sub-array `a[k]` are accessed between them. Because the data at `a[i][j+1]` may be removed from the cache memory after they have been executed, MDGLIA moves `a[i][j+1]` immediately ahead of `a[k][l]`, as shown in Figure 3(b).

Our goal in this paper is to provide algorithms for aggregating array references and analyze effective of our algorithm in detail. In our previous paper [24], we showed that our algorithm is effective for SPEC CPU 2000 benchmark on an Intel Core i5 processor, which is the processor used by popular desktop computers. In this paper, we demonstrate the effects of our algorithm on other grade CPUs, specifically, Xeon and Core2duo, which have larger and smaller cache memories, respectively, than Core i5. We implemented MDGLIA and applied it to two kinds of benchmarks: matrix multiplication and SPEC CPU 2000. For the matrix multiplication, we conducted experiments on an Intel Xeon processor possessing large cache memories in order to determine how our aggregation approach reduces cache misses. Conversely, for the SPEC CPU 2000 benchmarks, we conducted experiments on an Intel Core2Duo processor possessing small cache memories similar to the CPUs used in embedded systems in order to examine how our code motion reduces cache misses in a real program. The results indicate that our algorithm is effective in various types of computers.

The advantages of our algorithm are summarized as follows:

- MDGLIA moves array references in order to remove redundancy and reduce the number of cache misses while considering array dimensions.
- MDGLIA suppresses spills without reducing the effect of aggregation to prevent cache misses.
- We implemented MDGLIA in a compiler and conducted experiments for two benchmarks on two CPUs.

The rest of this paper is organized as follows: Section 2 summarizes related works in this area. Section 3 presents preliminary definitions needed to explain our algorithm, and outlines laze code motion (LCM)–a variant of PRE [15]. Section 4 outlines our proposed MDGLIA algorithm that is extended from LCM. Section 5 presents and analyzes experimental results that demonstrate the effectiveness of MDGLIA. Section 6 concludes this paper.

2. Related Work

2.1. Cache Optimization

Data in memory that has been used once tends to get used again in the near future, and the surrounding data also tends to be used in much of the program. These two phenomena are called temporal locality and spatial locality, respectively, and are used to improve the execution efficiency of programs via cache memories. Popular techniques for enhancing the localities are due to transforming loop structures [1, 2]. Although these techniques often significantly improve the execution efficiency of a program, their application tend to be limited to specific control structures such as simple loops. However, our algorithm is based on

global code motion that does not change the control structure of a program, and therefore can be applied to any program. Furthermore, our algorithm and the abovementioned loop transformation based techniques can both be applied to a program at the same time to cooperatively improve the program.

Techniques that use prefetchers are also frequently used to improve the utilization of cache memories. The basic idea underlying these techniques is prefetching of data by transferring that data into a cache memory before it is used by programs. There are two prefetch levels: software and hardware. Software prefetch can be achieved by explicitly inserting fetch instructions; it is particularly effective for loops with array accesses. Hardware prefetch implicitly achieves effective prefetch by exploiting spatial locality. According to Smith [22], there are three variants of *one block lookahead* (OBL): The first prefetches line $l + 1$ whenever line l is being accessed (*always prefetch*). The second prefetches line $l + 1$ whenever access to line l results in a cache miss (*prefetch on miss*). The third prefetches line $l + 1$ only for the first access to line l (*tagged prefetch*). Hardware prefetch does not need a change in the input program, but is effective only for good spatial locality. Non-sequential memory access significantly decreases execution performance. Our algorithm not only removes redundant array references, but also improves spatial locality by moving array references. In other words, removing redundancies and making array references continuous, reduces the number of memory accesses and improves spatial locality. Consequently, our algorithm can enhance the effect of prefetch, especially hardware prefetch.

Some algorithms improve cache efficiency based on data-layout. Cache-conscious data placement (CCDP) reduces cache conflict misses by considering data-layout [4]. CCDP uses a *temporal relationship graph* (TRG), in which nodes represent objects (e.g., functions, arrays, and global variables) to be placed in the data cache. The edges between the objects represent the estimated number of cache misses that would occur if the two objects were mapped to the same cache set. A compiler assigns addresses to the objects based on a conflict cost metric calculated for the TRG in order to minimize cache conflict misses.

Although CCDP can reduce cache conflict misses for a processor core with a single execution context, it loses much of its benefit in a multithreaded environment because inter-thread conflicts are not deterministic. Sarkar and Tullsen proposed an algorithm that extends CCDP to multithreaded architectures [21]. In their algorithm, the extra cost incurred from the sharing of objects by threads in cache blocks is added to each TRG edge as weight. Like CCDP, the compiler assigns addresses to the objects using a TRG in order to minimize the cost of cache conflict misses. Ishitobi et al. proposed an algorithm based on object layout that reduces the energy consumption of on-chip memory by considering memory allocation on a processor that has cache memory and scratchpad memory [13]. Their algorithm partitions the memory into cacheable region, scratchpad region, and non-cacheable regions in order to minimize the total energy consumption and the number of cache misses. SwiftArray uses Hilbert space-filling curve instead of row-major order as the data-layout to improve the spatial locality of multidimensional arrays [9].

The above approaches focus on data allocation layout in memory but not the reordering of the execution code. The allocation layout information can be used synergistically to improve our algorithm by specifying effective target array references. That is, Ishitobi et al.'s algorithm can be combined with our algorithm by first applying their algorithm, and then using our algorithm to move array references whose reference data is allocated in a cacheable region. This combination improves utilization of cache memory while reducing the energy consumed. If our algorithm is utilized on a computer that uses SwiftArray, our algorithm may not be effective because our algorithm assumes that array data is allocated in row-major order as mentioned in Section 3.1. Thus, in order to integrate our algorithm with their data-layout, our aggregation targets should be modified to consider the layout of the data.

2.2. Removing Redundant Expressions

The original PRE algorithm that uses bidirectional data-flow analysis was proposed by Morel and Renvoise [18]. This algorithm can eliminate some redundancies and move loop-invariant expressions out of loops, but some redundant expressions are not removed because the algorithm does not insert expressions at nodes where downsafe are not satisfied. Dhamdhere extended this algorithm to insert expressions on edges [7], and Dhamdhere and Patil also proposed another algorithm that removes redundancies based on unidirectional data-flow analysis [8]. Bodik et al. proposed the removal of all redundant expressions by copying certain parts of the program [3]. Kawahito et al. proposed an algorithm that removes partially redundant load and store instructions based on extended PRE [14]. These algorithms remove redundant expressions, but they are not capable of reducing the number of cache misses.

Speculative code motion, another algorithm based on code motion, moves load instructions out of loops, but the movement may introduce some new computations [16]. This algorithm has to recognize loop structures whereas our algorithm can be applied to an entire program without recognizing them, because our algorithm is based on PRE.

3. Background

In this section, we give the definitions of the program representations assumed by our algorithm, and outline LCM (lazy code motion).

3.1. Preliminaries

We assume that MDGLIA is applied to the intermediate representation converted from a source program, which is represented as a sequence of statements with at most one operator or function. Some statements include load and store instructions with memory access, which is described as an array reference, e.g., $a[i][j]$ with address a and indexes i and j. The statement loading the data at memory location $a[i][j]$ into a virtual register x is expressed as assignment statement $x = a[i][j]$, which we call a *load statement*. The right-hand side of an assignment statement is called an expression. Further, a statement that stores the data in a virtual register x to a memory location $a[i][j]$ is expressed as $a[i][j] = x$, which we call a *store statement*.

We assume that the memory access appears solely in the assignment. That is, a procedure call $f(a[i][j])$ is split into two statements; e.g., $t = a[i][j]$ and $f(t)$. Similarly, a store statement with a nested array reference $a[i][j] = a[b[i]][k]$ is split into three statements; e.g., $t = b[i]$, $t_0 = a[t][k]$ and $a[i][j] = t_0$. Moreover, the load statement includes a temporary virtual register instead of an original virtual register in the left-hand side. For example, the load statement $i = a[i][j]$ is split into two statements, such as $t = a[i][j]$ and $i = t$, by introducing a temporary virtual register t. We assume that any arrays are laid out in row-major order on the memory, such as arrays in C, which means that the leftmost index corresponds to the highest dimension of the array.

We assume that a CFG (control flow graph) has been built for each program. A CFG is a graph structure represented by a quadruple ($\mathbf{N}$,$\mathbf{E}$,$\mathbf{s}$,$\mathbf{e}$), where $\mathbf{N}$ is a set of nodes with a single statement, $\mathbf{E}$ (denoted by $\mathbf{N} \times \mathbf{N}$) is a set of edges, $\mathbf{s}$ is a start node with an empty statement, and $\mathbf{e}$ is an end node with an empty statement. A given edge is expressed as $(m, n) \in \mathbf{E}$, where m is called a predecessor of n, and n is called a successor of m. In general, there are several predecessors and successors for a node, because of the nondeterministic branching structure of a CFG. Hence, the sets of predecessors and successors of node n are denoted by node sets $pred(n)$ and $succ(n)$, respectively.

A CFG assumes that any *critical edge* that leads from a node with more than one successor to a node with more than one predecessor has been removed and replaced with a synthesized node, because critical edges can block effective code motion. In fact, this assumption has been adopted for the same reason in many traditional algorithms based on code motion [15].

3.2. Lazy Code Motion

PRE removes redundant expressions by inserting some expressions; however, the process of insertion and removal tends to lengthen the live-ranges of variables carrying loaded values to their utilization, leading to spills in the register allocation phase [6, 20]. To address this problem, LCM consists of the first code motion hoisting expressions as early as possible and the second code motion delaying them as late as possible. The first code motion eliminates all removable expressions, and the second one minimizes the live-ranges of variables.

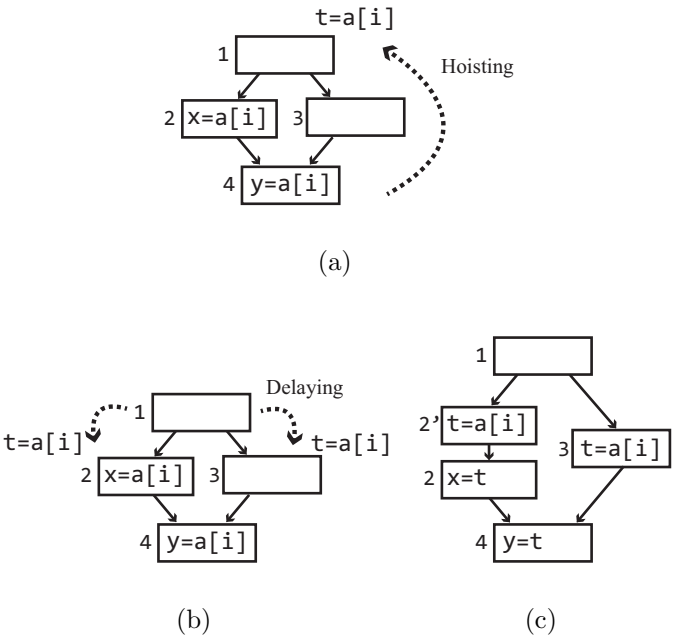

Figure 4: Code motions of LCM: (a) Hoisting expressions. (b) Delaying expressions. (c) Result of translation of LCM.

For example, consider an array reference `a[i]` at Node 4 in Figure 4(a). This array reference is partially redundant because it is redundant on a path through Node 2, whereas it is not on another path through Node 3. To remove this redundancy, LCM first determines the CFG nodes at which array references can be inserted. If an array reference is inserted at Node 1, the redundant array reference can be removed by replacing it with the temporary `t` introduced. Note here that, the live-range of `t` is lengthened; therefore, LCM delays the insertion node, as shown in Figure 4(b). Finally, LCM inserts the array references at Nodes 2 and 3, and then replaces the original `a[i]` with `t`, as shown in Figure 4(c).

These code motions have to satisfy two kinds of *safeties*: *down-safety* and *up-safety*. Down-safety is used to ensure that LCM does not lengthen any execution paths. It is represented by the predicate $DownSafe(n)$ that denotes that there are occurrences of the expression on all sub-paths following the node n. Up-safety is used to ensure that there are some paths where the number of expressions is reduced by the insertions and removals. Up-safety is represented by the predicate $UpSafe(n)$ that denotes that there are some occurrences of the expression on all sub-paths leading to the node n. These safeties are defined under the condition of *transparency* that ensures that the value of an expression does not change at the program points of concern. LCM represents this condition using the predicate *Transp*. Because up-safety is one of the most important predicates in our algorithm, we present its formal definition below.

$$UpSafe(n) \stackrel{def}{\Leftrightarrow} (n \neq \mathbf{s}) \wedge \prod_{m \in pred(n)} Comp(m) \vee (Transp(m) \wedge UpSafe(m)) \quad (1)$$

where the predicate $Comp(n)$ denotes that node n includes the same expression.

For example, as shown in Figure 4(a), the *DownSafe* of Nodes 1, 2, and 3 is *true* because there is no modification for `i` nor `a[i]` in Nodes 2, 3, and 4. In contrast, the *UpSafe* of all nodes is *false*. Because no occurrence of the array reference on a path from Node 1 to Node 3, $UpSafe(4)$ is *false*. The *UpSafe* of the other nodes is also obviously *false*.

LCM determines two kinds of insertion nodes based on the predicates *Earliest* and *Latest*, which determine the insertion points in the two code motions mentioned above, respectively. $Earliest(n)$ denotes that node n is the closest to $\mathbf{s}$ of the nodes m satisfying $DownSafe(m)$. $Latest(n)$ denotes that node n, all paths to which include some nodes satisfying *Earliest*, is the node

closest to node c satisfying $Comp(c)$, and any node on the path from n to c does not satisfy $Comp$. $Latest(n)$ is defined on the basis of maximal fixed points of the data-flow equation for the predicate $Delayed(n)$ that denotes that the expression can be delayed until the exit of node n.

Consider `a[i]` in Figure 4 (a). At the beginning, LCM decides that $Earliest(1)$ is *true* because $DownSafe(1)$ is *true*, and this node is the closest to the start node. LCM subsequently delays the insertion points through the decisions of *Delayed* and *Latest*, which result in *true* for $Comp(2)$, *false* for $Delayed(4)$, and *true* for *Latest* at Nodes 2 and 3.

Here, because *Delayed* is also one of the most important predicates for our algorithm, we present the formal definition as follows:

$$Delayed(n) \stackrel{def}{\Leftrightarrow} Earliest(n) \vee (n \neq \mathtt{s}) \ \wedge \prod_{m \in pred(n)} \neg Comp(m) \wedge Delayed(m)$$

LCM inserts expressions at the entry of nodes n satisfying the predicate $Insert(n)$ that denotes that n is one of the nodes satisfying *Latest*. Note that LCM does not insert any expression without reducing the number of expressions on some paths, because such insertions are unnecessary. Therefore, $Insert(n)$ is defined as $Latest(n) \wedge \neg Isolated(n)$ on the basis of the predicate $Isolated(n)$ that denotes that the insertion at n enables the removal of no expression other than the original one. After the insertion at the entry points of nodes n satisfying $Insert(n)$, each of the same expression at node n satisfying $\neg(Latest(n) \wedge Isolated(n))$ is replaced by the temporary variable holding the value of the expression.

4. Aggregating Array References

Similar to LCM, MDGLIA checks *DownSafe*, *UpSafe*, *Earliest*, *Delayed*, and *Latest* to determine the nodes to move an array reference ar. In the delaying, MDGLIA checks start addresses and the number of corresponding indexes of array references for aggregation while keeping the order of accesses to arrays.

4.1. Local Properties

MDGLIA defines the local properties $SameAddr$, $Transp_e$, $Transp_{Addr}$, and *isSame* to check addresses. *SameAddr* and *isSame* represent equalities of start addresses and entire expressions, respectively. $SameAddr(n)$ denotes that n contains a load statement referring to the same array reference as ar. $isSame(n)$, corresponding to *Comp* of LCM, denotes that $SameAddr(n)$ is *true* and the indexes are the same as ar. $Transp_e(n)$, which is defined as transparency, denotes that there is no modification to ar and no store statement to the array referred to by ar in n. $Transp_{Addr}$ denotes that there is neither modification to ar nor reference to arrays different from ar.

To determine these predicates, we use predicates *rhs*, *Load*, *TopAddr*, *Store*, *Def*, and *Var*. $rhs(n)$ returns the right-hand side of a statement at node n. $Load(n)$ denotes that node n includes a load statement. $TopAddr(ar)$ returns the start address of ar if ar is an array; otherwise, it returns $\perp$. $Store(n)$ denotes that node n includes a store statement that accesses the same array as ar. $Def(n)$ gives a variable defined at node n. $Var(ar)$ gives the set of variables used in ar. The local properties are defined as follows:

$$\begin{aligned}
SameAddr(n) &\stackrel{def}{\Leftrightarrow} Load(n) \wedge (TopAddr(rhs(n)) = TopAddr(ar)) \\
Transp_e(n) &\stackrel{def}{\Leftrightarrow} Def(n) \notin Var(ar) \wedge \neg Store(n) \\
Transp_{Addr}(n) &\stackrel{def}{\Leftrightarrow} Transp_e(n) \wedge (\neg Load(n) \vee SameAddr(n)) \\
isSame(n) &\stackrel{def}{\Leftrightarrow} rhs(n) \neq \perp \wedge rhs(n) = ar
\end{aligned}$$

4.2. Modified Global Properties

MDGLIA modifies *UpSafe* and *Delayed* in LCM.

4.2.1. Extending UpSafe

$UpSafe(n)$ is modified to denote that there are some *array references whose start addresses are the same as ar* on all sub-paths leading to node n. This predicate is formally defined as follows:

$$UpSafe(n) \overset{def}{\Leftrightarrow} (n \neq \mathbf{s}) \wedge \prod_{m \in pred(n)} SameAddr(m) \vee (Transp_e(m) \wedge UpSafe(m)) \quad (2)$$

By this modification, it is able to analyze the access order of arrays. However, it *speculatively* moves array references although LCM does not perform speculative movement.

Consider application of our algorithm to `a[i][j+1]` in Figure 5(a). Because an array reference `a[i][j]` is executed at Node 1, $UpSafe(2)$ is *true*, and therefore, $Earliest(2)$ is *true*. Consequently, it leads $Delayed(2)$ is also *true*, but $Delayed(3)$ becomes *false* because $Transp_{Addr}(2)$ is *false*. As a result, $Latest(2)$ becomes *true*, and therefore, `a[i][j+1]` is moved immediately before Node 2 as shown in Figure 5(b). Note here that `a[i][j+1]` does not originally exist on the path through Node 3 in Figure 5(a). That is, this insertion is speculative. Speculative code motion may increase the number of expressions on some paths; however, if the speculatively inserted array references cause cache misses, the execution efficiency would be significantly decreased.

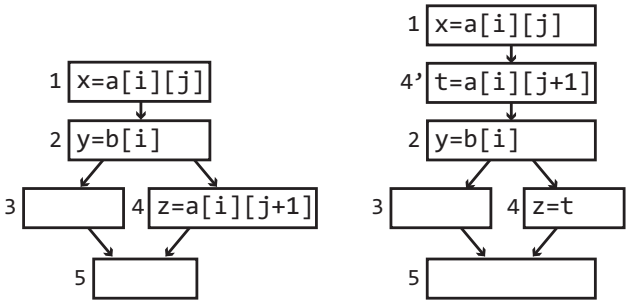

Figure 5: Speculative code motion: (a) Original code. (b) Moving array reference, not satisfying down-safety.

In general, speculatively moving load statements is unsafe; it may cause out-of-bounds exceptions. If the objective codes are executed on processors supporting the exception handler, there is no problem. In fact, most modern architectures can resolve the problem by suppressing the exceptions [10, 17], but for architectures that do not support it, speculative code motion should be forbidden using equation (1).

4.2.2. Extending Delayed

Delayed is modified to check whether the *keep-order condition* and *keep-dimension condition* are satisfied. We describe the details of these conditions in this section.

Keep-order Condition

The keep-order condition guarantees that there is no reference to the array different from ar at n after the preceding reference to the same array as ar. The keep-order condition is represented by $keepOrder$, and is defined as follows:

$$partialUpSafe(n) \overset{def}{\Leftrightarrow} \sum_{m \in pred(n)} UpSafe(m)$$

$$keepOrder(n) \overset{def}{\Leftrightarrow} \neg partialUpSafe(n) \vee Transp_{Addr}(n)$$

Figure 6(a) demonstrates the effect of aggregation considering the keep-order condition. Consider moving `a[k][l]` at Node 6. Nodes 4 and 6 contain array references that reference array `a`, but there is an array reference that references array `b` between them. To move `a[k][l]` immediately ahead of Node 5, MDGLIA determines *Earliest* by first checking *UpSafe* and *DownSafe*. Because there is no `a[k][l]` on the path through Node 2, $DownSafe(1)$ and $UpSafe(1)$ are *false*. These results cause $Earliest(1)$ to be *false*. On the other hand, another path through Node 3 has the same reference at Node 6. Hence, $DownSafe(3)$ and $Earliest(3)$ are *true*, which causes $Delayed(3)$ to be *true*. At each node from Node 3, MDGLIA checks whether the keep-order condition is satisfied. Node 4 has an array reference that references the same array as `a[k][l]`, but Node 5 has another array reference that references a different array. Hence, $SameAddr(4)$ is *true*, and then $UpSafe(5)$ is *true*, but $SameAddr(5)$ is *false*. These cause $keepOrder(5)$ and $Delayed(6)$ to be *false*. Eventually, $Latest(5)$ and $Insert(5)$ are *true*; therefore, MDGLIA inserts an array reference immediately before Node 5 and removes the original expression, as shown in Figure 6(b).

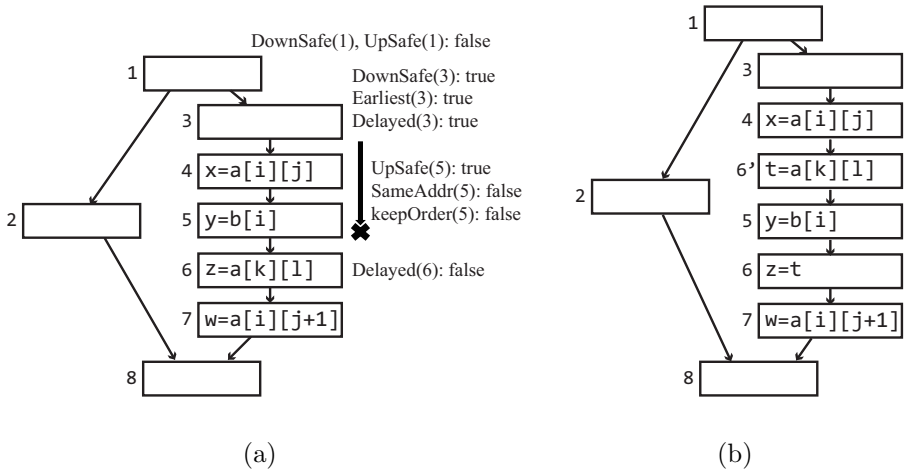

Figure 6: Effectiveness of extending *UpSafe*: (a) Result of computing data-flow equations to move `a[k][l]` at Node 6. (b) Result of moving the array reference.

Keep-dimension Condition

The keep-dimension condition contributes to the aggregation of references to closer addresses in the same array by keeping the number of corresponding indexes from decreasing. This condition uses two kinds of predicates to preserve the number: *nCI* (number of corresponding indexes) and *pnCI* (propagated *nCI*).

To represent closeness on the main memory between references to an array, we define *nCI* as the number of higher indexes corresponding with ar. $nCI(n)$ checks whether each index of the array reference at n is same as corresponding index of ar, in succession from the left, whenever n contains a reference to the same array as ar. This predicate is formally defined as follows:

Definition 1 (*nCI*)**.** *Assume that* ar *is* $a[i_1][i_2]...[i_k]...[i_n]$*, and* r *is* $b[j_1][j_2]...[j_k]...[j_m]$*. Then,*

$$nCI(r) \overset{def}{\Leftrightarrow} \begin{cases} k & if\ a = b \wedge i_l = j_l \wedge \\ & \quad (k = n \vee k = m \vee \\ & \quad\quad i_{k+1} \neq j_{k+1}) \\ & \quad where\, \forall l \in \{1, ..., k\} \\ 0 & otherwise \end{cases}$$

The condition means that if their start addresses are the same, these indexes are checked to determine whether the kth index is the same until the last index.

The *pnCI* of n denotes the number of indexes to the reference that has the most indexes that are the same as ar on all execution paths to the exit of n from the preceding node where *SameAddr* is *true*. We represent the *pnCI* of n as $pnCI(n)$, which can be defined by the data-flow equation based on *nCI* as follows:

$$pnCI(n) \overset{def}{\Leftrightarrow} \begin{cases} 0 & if\ n = \mathbf{s} \\ nCI(n) & if\ SameAddr(n) \\ Max(\{pnCI(m) \mid & \\ \quad m \in pred(n)\}) & otherwise \end{cases}$$

The equation can be solved iteratively as well as using typical data-flow analysis. If node n includes a reference to the same start address as ar, then $pnCI(n)$ is $nCI(n)$. Otherwise, $pnCI(n)$ is the maximal value of the *pnCI* of $pred(n)$, excluding the start node. Thus, *pnCI* represents the advantage of moving ar to n for cache-hits.

The keep-dimension condition is represented by the predicate *keepDimension*, and is defined using *partialUpSafe* and *pnCI* as follows:

$$keepDimension(n) \overset{def}{\Leftrightarrow} \neg partialUpSafe(n) \vee \prod_{m \in pred(n)} pnCI(n) \geq pnCI(m)$$

This predicate denotes that n does not include any reference with *nCI* that is less than the preceding references.

Delaying Under Keep-order and Keep-dimension Conditions

MDGLIA checks whether the keep-order and keep-dimension conditions are satisfied in addition to *Delayed* of LCM. Once these conditions are introduced, *Delayed* is simply changed as follows:

$$Delayed(n) \stackrel{def}{\Leftrightarrow} Earliest(n) \vee (n \neq \mathbf{s}) \ \wedge \prod_{m \in pred(n)} \neg isSame(m) \wedge keepOrder(m) \wedge keepDimension(m) \wedge Delayed(m)$$

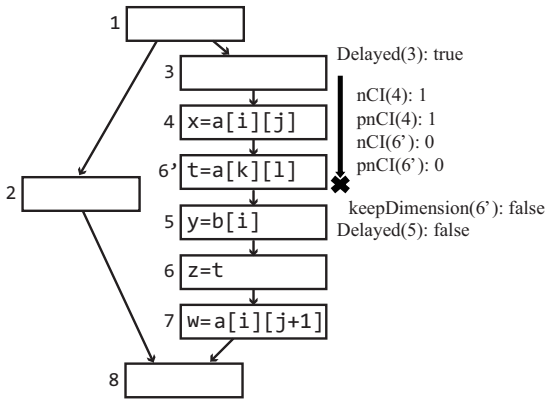

Figure 7: Computing the closeness of each address of `a[i][j+1]` at Node 4 and *Delayed*.

Figure 7 shows the results of data-flow analysis of MDGLIA when it is applied to `a[i][j+1]` at Node 7 in Figure 6 (b). Nodes 4 and 7 contain array references that reference sub-array `a[i]`, but there is an array reference that references sub-array `a[k]` between them. To move `a[i][j+1]` immediately before Node 6', MDGLIA determines *Earliest* first. Because *Earliest*(3) is *true* as well as the case of moving `a[k][l]` in Figure 6(a), *Delayed*(3) is also *true*. At each node from Node 3, MDGLIA checks whether the keep-dimension condition is satisfied. Although *nCI* and *pnCI* of Node 4 are one, *nCI* and *pnCI* of the successor Node 6' are zero; therefore, *keepDimension*(6') and *Delayed*(5) are *false*. Eventually, *Latest*(6') and *Insert*(6') are *true*, which results in MDGLIA inserting an array reference immediately before Node 6' and removing the original expression, as shown in Figure 3 (b).

Finally, to aid understanding of the overall algorithm, we present formal definitions of all global predicates, apart from *UpSafe*, *keepOrder*, *keepDimension*, and *Delayed*, in Figure 8.

4.3. Application to the Overall Program

MDGLIA traverses CFG in *topological sort order*. When an array reference is found during the traversal, MDGLIA determines local/global properties for it. This analysis style is called *demand-driven* analysis. Traditional PREs are designed as an exhaustive analysis that determines the redundancy of all expressions using bit vectors and checking lexical equality. In general, application of PRE based approaches exposes new redundant expressions. This effect is called the *second order-effect*. Reflecting as many of them as possible results in the removal of more redundant expressions. However the reflection of all second order-effects requires iterative applications of PRE that are costly because PRE is traditionally designed based on exhaustive data-flow analysis [19]. On the other hand, demand-driven application of MDGLIA to the entire program in topological sort order facilitates efficient capture of the second order-effects [23]. Further, the manner in which MDGLIA is applied eliminates unnecessary code motion because array references are moved one by one.

In Figure 9(a), Nodes 2 and 3 might expel data that may be used later from cache memory. If MDGLIA exhaustively moved all array references at the same time, both references `a[i][j+1]` and `a[k][l+1]` would be moved, as shown in Figure 9(b). Although the code motion can address the above problem, it enhances the register pressure of the temporary variable to hold the value of `a[k][l+1]` because of the unnecessary code motion of `a[k][l+1]`. In contrast, MDGLIA traverses Nodes 1, 2, 3, and 4 in this order. Considering the case where Node 3 is visited, `a[i][j+1]` is moved to the entry of Node 2 based on the demand-driven application manner as shown in Figure 9(c).

$$DownSafe(n) \overset{def}{\Leftrightarrow} (n \neq \mathbf{e}) \wedge (isSame(n) \vee Transp_e(n) \wedge \prod_{m \in succ(n)} DownSafe(m))$$

$$Safe(n) \overset{def}{\Leftrightarrow} UpSafe(n) \vee DownSafe(n)$$

$$Earliest(n) \overset{def}{\Leftrightarrow} Safe(n) \wedge ((n = \mathbf{s}) \vee \sum_{m \in pred(n)} \neg Transp_e(m) \vee \neg Safe(m))$$

$$Latest(n) \overset{def}{\Leftrightarrow} Delayed(n) \wedge (isSame(n) \vee \neg \prod_{m \in succ(n)} Delayed(m))$$

$$Isolated(n) \overset{def}{\Leftrightarrow} \prod_{m \in succ(n)} (Latest(m) \vee \neg isSame(m) \wedge Isolated(m))$$

$$Insert(n) \overset{def}{\Leftrightarrow} Latest(n) \wedge \neg Isolated(n)$$

$$Replace(n) \overset{def}{\Leftrightarrow} isSame(n) \ \wedge \neg(Latest(n) \wedge Isolated(n))$$

Figure 8: Data-flow equations used in MDGLIA.

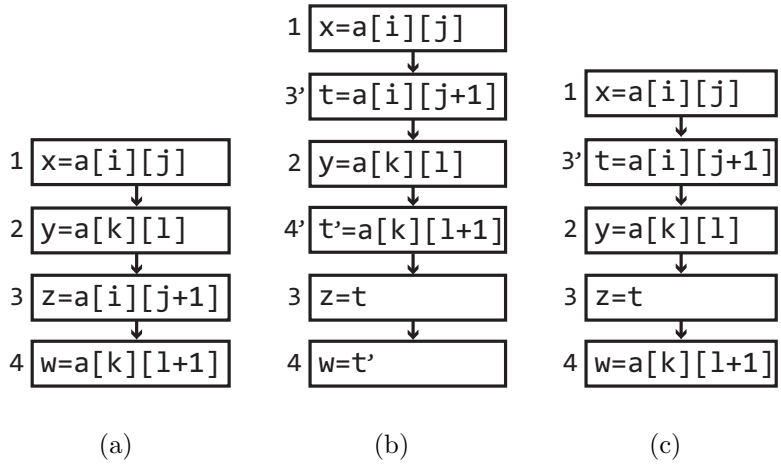

Figure 9: Preserving unnecessary code motion: (a) Original code. (b) Result of applying exhaustive analysis version of MDGLIA to `a[i+1]`. (c) Result of the applying demand-driven style of MDGLIA to `a[i+1]`.

5. Experimental Results

We implemented MDGLIA as a low-level intermediate representation converter in a COINS compiler[1], which is a compiler infrastructure supported by the open-source community.

We conducted a matrix multiplication experiment (Section 5.1) and a SPEC CPU 2000 benchmark experiment (Section 5.2). In the former experiment, we investigated the operation of our aggregation on an Intel Xeon computer that has large cache memories. In the latter experiment, we examined how well our speculative code motion reduces the number of cache misses on an Intel Core2Duo computer with a small amount of cache memory, similar to the processors used in embedded systems.

5.1. Effectiveness of Aggregation

We used the matrix multiplication program shown in Figure 10 in this experiment. We assumed that a scalar replacement was applied to

[1] http://coins-compiler.sourceforge.jp/

Matrix Multiplying with Scalar Replacement
double A, double B, double C,
$COL = ROW = 500$
1: **for** $(i = 0; i < COL; i{+}{+})$
2: **for** $(j = 0; j < ROW; j{+}{+})$
3: $tmp = 0.0$
4: **for** $(k = 0; k < ROW; k{+}{+})$
5: $tmp{+} = A[i][k] * B[k][j]$
6: $C[i][j] = tmp$

Figure 10: Matrix multiplication program following application of scalar replacement.

Tiling
1: **for** $(i_0 = 0; i_0 < COL; i_0{+}{=}8)$
2: **for** $(j_0 = 0; j_0 < ROW; j_0{+}{=}8)$
3: $min_i = min(i_0 + 7, COL)$
4: **for** $(i = i_0; i < min_i; i{+}{+})$
5: $min_j = min(j_0 + 7, ROW)$
6: **for** $(j = j_0; j < min_j; j{+}{+})$
7: $tmp = 0.0$
8: **for** $(k = 0; k < ROW; k{+}{+})$
9: $tmp{+} = A[i][k] * B[k][j]$
10: $C[i][j] = tmp$

Figure 11: Result of Tiling.

the program [5].

We used an x64 machine with an Intel Xeon(R) CPU E5-1660 3.30GHz CPU and Debian OS. The perf utility was employed to determine how effectively misses of L1 data cache (L1D) and the last level cache memory (LLC) can be reduced by our aggregation. The cache memory parameters for the x64 machine are shown in Table 1.

Table 1: Cache memory parameters

Parameters	L1D	L2	L3
Total size (KB)	32	256	15,360
Line size (bytes)	64	64	64
Number of cache line	512	4,096	245,760
Associativity	8	8	20

To estimate effect of our aggregation, we discuss four results: comparisons of 1) keep-order condition and 2) keep-dimension condition with loop tiling without hardware prefetch, 3) comparison of MDGLIA with LCM, and 4) comparison of MDGLIA with loop tiling using hardware prefetch.

5.1.1. Aggregation Under Keep-order Condition

To investigate how our aggregation without considering array dimensions reduces the number of cache misses, we adopted the following two kinds of optimization settings:

Tiling, application of scalar replacement and loop tiling.

MDGLIA-k-Unroll, application of scalar replacement, loop tiling, unrolling the k-loop, and MDGLIA.

Applying these optimization settings result programs outlined in Figure 11 and Figure 12.

Loop tiling inserts two new loops (i_0- and j_0-loops) based on the size of each cache line. Because the size of each cache line of the Xeon processor is 64, and the size of each element in matrices A and B is eight, we add i_0 and j_0 eight for each iteration of the i_0- and j_0-loops.

The execution time and the number of cache misses in the case where the optimizations are applied without hardware prefetch are listed in Table 2.

Table 2: Result of execution time and cache misses obtained by Tiling and MDGLIA-k-Unroll without hardware prefetch

	A.Tiling	B.MDGLIA-k-Unroll	(A-B)/A
time	0.265	0.258	2.8%
L1D miss	576,314,189	768,844,837	-33.4%
LLC miss	2,083,135	2,078,436	0.2%

Loop unrolling can reduce the number of conditional jumps; therefore, it improves execution efficiency. However, the L1D misses for MDGLIA-k-Unroll are greater than that for Tiling because the number of spills is increased as shown in Table 3. Once a variable is spilled, additional load/store statements are inserted around references of the variable; therefore, L1D misses may also occur for the inserted load/store statements. Note here that the effect of the spills on cache miss depends on the locations where the load/store statements are inserted. Although MDGLIA-k-Unroll increased only one spill, the extra load/store statements could be inserted into the innermost loop.

Applying Tiling, unrolling the k - loop seven times, and MDGLIA

```
 1: for (i_0 = 0; i_0 < COL; i_0+=8)
 2:   for (j_0 = 0; j_0 < ROW; j_0+=8)
 3:     min_i = min(i_0 + 7, COL)
 4:     for (i = i_0; i < min_i; i++)
 5:       min_j = min(j_0 + 7, ROW)
 6:       for (j = j_0; j < min_j; j++)
 7:         tmp = 0.0
 8:         for (k = 0; k < ROW;)
 9:           if(k + 7 < ROW)
10:             a_1 = A[i][k]; a_2 = A[i][k + 1]; a_3 = A[i][k + 2]; a_4 = A[i][k + 3];
11:             a_5 = A[i][k + 4]; a_6 = A[i][k + 5]; a_7 = A[i][k + 6]; a_8 = A[i][k + 7];
12:             b_1 = B[k][j]; b_2 = B[k + 1][j]; b_3 = B[k + 2][j]; b_4 = B[k + 3][j];
13:             b_5 = B[k + 4][j]; b_6 = B[k + 5][j]; b_7 = B[k + 6][j]; b_8 = B[k + 7][j]
14:             tmp+ = a_1 * b_1 + a_2 * b_2 + a_3 * b_3 + a_4 * b_4 + a_5 * b_5 + a_6 * b_6 + a_7 * b_7 + a_8 * b_8
15:             k+=8
16:           else
17:             tmp+ = A[i][k] * B[k][j]
18:             k++
19:         C[i][j] = tmp
```

Figure 12: Result of MDGLIA-k-Unroll.

Each statement in the innermost loop will be executed $500 \times 500 \times 500$ times; thus, the increased spills can worsen the results. The LLC miss results are virtually similar. This is because of the large sizes of the L2 cache memory and the LLC ; we suppose these cache memories record data missed in L1D. In addition, loop tiling can also address the same problem as MDGLIA-k-Unroll.

Table 3: Number of spills occurred by Tiling and MDGLIA-k-Unroll

	A.Tiling	B.MDGLIA-k-Unroll
spills	3	4

5.1.2. Aggregation Under Keep-dimension Condition

To examine how aggregation considering array dimensions reduces the number of cache misses, we used the following optimization setting:

MDGLIA-j-Unroll, application of scalar replacement, loop tiling, unroll-and-jam to the j-loop, unrolling the k-loop, and MDGLIA.

Unrolling the j-loop reveals redundant array references to array A, as shown in Figure 13; subsequent application of MDGLIA results in the program shown in Figure 14.

$$
\begin{aligned}
&tmp11+=A[i][k] * B[k][j]\\
&tmp12+=A[i][k+1] * B[k+1][j]\\
&tmp13+=A[i][k+2] * B[k+2][j]\\
&tmp14+=A[i][k+3] * B[k+3][j]\\
&tmp15+=A[i][k+4] * B[k+4][j]\\
&tmp16+=A[i][k+5] * B[k+5][j]\\
&tmp17+=A[i][k+6] * B[k+6][j]\\
&tmp18+=A[i][k+7] * B[k+7][j]\\
&tmp11+=A[i][k+8] * B[k+8][j]\\
&tmp21+=A[i][k] * B[k][j+1]\\
&tmp22+=A[i][k+1] * B[k+1][j+1]\\
&tmp23+=A[i][k+2] * B[k+2][j+1]\\
&tmp24+=A[i][k+3] * B[k+3][j+1]\\
&tmp25+=A[i][k+4] * B[k+4][j+1]\\
&tmp26+=A[i][k+5] * B[k+5][j+1]\\
&tmp27+=A[i][k+6] * B[k+6][j+1]\\
&tmp28+=A[i][k+7] * B[k+7][j+1]\\
&tmp21+=A[i][k+8] * B[k+8][j+1]
\end{aligned}
$$

Figure 13: Revealing redundant array references loading data of array A by unroll-and-jam.

The execution time and cache misses of Tiling and MDGLIA-j-Unroll are listed in Table 4. The execution time of MDGLIA-j-Unroll is better than that of Tiling about 40% because MDGLIA

Invoking unroll - and - jam to j - loop , Unrolling k - loop , and applying MDGLIA

```
 1: for (i0 = 0; i0 < COL; i0+=8)
 2:   for (j0 = 0; j0 < ROW; j0+=8)
 3:     min_i = min(i0 + 7, COL)
 4:     for (i = i0; i < min_i; i++)
 5:       min_j = min(j0 + 7, ROW)
 6:       for (j = j0; j < min_j;)
 7:         if(j + 1 < ROW)
 8:           tmp1 = 0.0
 9:           tmp2 = 0.0
10:           for (k = 0; k < ROW;)
11:             if(k + 7 < ROW)
12:               a1 = A[i][k]; a2 = A[i][k+1]; a3 = A[i][k+2]; a4 = A[i][k+3]; a5 = A[i][k+4];
13:               a6 = A[i][k + 5]; a7 = A[i][k + 6]; a8 = A[i][k + 7];
14:               b11 = B[k][j]; b21 = B[k][j + 1]; b12 = B[k + 1][j]; b22 = B[k + 1][j + 1]
15:               b13 = B[k + 2][j]; b23 = B[k + 2][j + 1]; b14 = B[k + 3][j]; b24 = B[k + 3][j + 1];
16:               b15 = B[k + 4][j]; b25 = B[k + 4][j + 1]; b16 = B[k + 5][j]; b26 = B[k + 5][j + 1];
17:               b17 = B[k + 6][j]; b27 = B[k + 6][j + 1]; b18 = B[k + 7][j]; b28 = B[k + 7][j + 1];
18:               tmp1+ = a1*b11+a2*b12+a3*b13+a4*b14+a5*b15+a6*b16+a7*b17+a8*b18
19:               tmp2+ = a1*b21+a2*b22+a3*b23+a4*b24+a5*b25+a6*b26+a7*b27+a8*b28
20:               k+=8
21:             else
22:               tmp1+ = A[i][k] * B[k][j]
23:               tmp2+ = A[i][k] * B[k][j + 1]
24:               k++
25:           C[i][j] = tmp1
26:           C[i][j + 1] = tmp2
27:           j+=2
28:         else
29:           // following code is same as lines 7–19 of Figure 12
```

Figure 14: Result of MDGLIA-j-Unroll.

Table 4: Result of execution time and cache misses obtained by each optimization without hardware prefetch

	A.Tiling	B.MDGLIA-j-Unroll	(A-C)/A
time	0.265	0.159	40.2%
L1D miss	576,314,189	729,992,458	-26.7%
LLC miss	2,083,135	2,338,954	-12.3%

can remove redundant array references revealed by loop unrolling. On the other hand, MDGLIA-j-Unroll increases the number of L1D and LLC misses. This is because some load/store statements that are inserted by spills can increase the misses as discussed in the previous section. Although LCM (lazy code motion) tries to reduce the number of spills by delaying insertion points as mentioned in Section 3.2, it can increase the number of spills [11]. In addition, MDGLIA tends to stop delaying earlier than LCM. For example, consider `a[i][j+1]` in Figure 15(a). The array reference is redundant; thus, it can be removed by LCM, as shown in Figure 15(b). On the other hand, this array reference is relevant for MDGLIA, which result in it moving, as shown in Figure 15(c). These observations show that MDGLIA may enhance register pressure more than LCM. If some store and load statements that occur as a result of spills are inserted between the array references to the same array, they may decrease the effects of MDGLIA.

The number of spills caused by Tiling and MDGLIA-j-Unroll are shown in Table 5. As shown, the number of spills caused by MDGLIA-j-Unroll is five times that of Tiling.

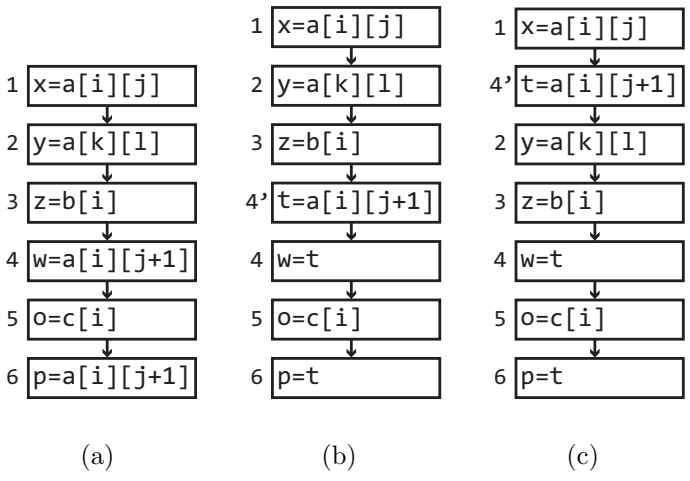

Figure 15: Difference in insertion point for `a[i][j+1]`: (a) Original code. (b) Result of applying LCM-MEM. (c) Result of applying MDGLIA.

Table 5: Number of spills occurred by Tiling and MDGLIA-j-Unroll

	A.Tiling	B.MDGLIA-j-Unroll	B/A
spills	3	15	5.0

In this section, we have discussed how our algorithm affects the execution times and L1D and LLC misses. In the following discussion, we will especially concentrate on the execution times and LLC misses because reducing the cache misses directly affects the execution time.

5.1.3. Comparison of Aggregation with Redundancy Elimination

Although loop unrolling and redundancy elimination improve execution efficiency, combining them results in an increase in the number of spills and cache misses. In order to simply check how our aggregation under keep-dimension condition reduces the number of cache misses, we used the following optimization setting:

LCM-j-Unroll, application of scalar replacement, loop tiling, unroll-and-jam to the j-loop, unrolling the k-loop, and LCM.

The resulting program for the LCM-j-Unroll is shown in Figure 16. The difference between LCM-j-Unroll and MDGLIA-j-Unroll lies in whether array references to array B are aggregated or not; MDGLIA-j-Unroll aggregates them, whereas LCM-j-Unroll does not.

The execution time and LLC misses for Tiling, MDGLIA-j-Unroll, and LCM-j-Unroll are shown in Table 6. Comparing MDGLIA-j-Unroll with LCM-j-Unroll, both the execution time and the number of LLC misses for MDGLIA-j-Unroll are better than those of LCM-j-Unroll. Further, comparing LCM-j-Unroll with Tiling, the execution time of LCM-j-Unroll is better than that of Tiling but the number of misses for LLC have increased. These results show that removing redundant load statements and aggregating array references significantly improve execution efficiency. In particular, aggregation considering array dimensions is effective for reducing LLC misses.

5.1.4. Effectiveness of the combination of Aggregation and Hardware Prefetch

As mentioned in Section 2, hardware prefetch is effective for good spatial locality. Because MDGLIA improves the locality, we examined how the combination of MDGLIA and hardware prefetch reduces the number of cache misses. The results for Tiling, MDGLIA-j-Unroll, and MDGLIA-j-7-Unroll (in which the j-loop is unrolled seven times) with hardware prefetch are shown in Table 7. Although the number of LLC missed for MDGLIA-j-Unroll is greater than that for Tiling, the rate of increase is about 4.3%. As shown in Table 4, the rate of increase for MDGLIA-j-Unroll without hardware prefetch is about 12.3%; therefore, MDGLIA with hardware prefetch is more effective than only MDGLIA. Comparing MDGLIA-j-7-Unroll with MDGLIA-j-Unroll, MDGLIA-j-7-Unroll reduces the number of LLC misses, but the execution time is increased. This is because unrolling the j-loop seven times can effectively utilize cache memory for loading data from array B, but it increases the number of spills, as shown in Table 8. Comparing MDGLIA-j-7-Unroll with Tiling, both the execution efficiency and the number of LLC misses are similar, but MDGLIA-j-7-Unroll increases the number of spills by a factor of 23 over loop tiling.

```
Invoking unroll - and - jam to j - loop , Unrolling k - loop , and applying LCM
 1: for (i0 = 0; i0 < COL; i0+=8)
 2:   for (j0 = 0; j0 < ROW; j0+=8)
 3:     min_i = min(i0 + 7, COL)
 4:     for (i = i0; i < min_i; i++)
 5:       min_j = min(j0 + 7, ROW)
 6:       for (j = j0; j < min_j;)
 7:         if(j + 1 < ROW)
 8:           tmp1 = 0.0
 9:           tmp2 = 0.0
10:           for (k = 0; k < ROW;)
11:             if(k + 7 < ROW)
12:               a1 = A[i][k]; a2 = A[i][k+1]; a3 = A[i][k+2]; a4 = A[i][k+3]; a5 = A[i][k+4];
13:               a6 = A[i][k + 5]; a7 = A[i][k + 6]; a8 = A[i][k + 7];
14:               b11 = B[k][j]; b12 = B[k + 1][j]; b13 = B[k + 2][j]; b14 = B[k + 3][j];
15:               b15 = B[k + 4][j]; b16 = B[k + 5][j]; b17 = B[k + 6][j]; b18 = B[k + 7][j];
16:               b21 = B[k][j+1]; b22 = B[k+1][j+1]; b23 = B[k+2][j+1]; b24 = B[k+3][j+1];
17:               b25 = B[k + 4][j + 1]; b26 = B[k + 5][j + 1]; b27 = B[k + 6][j + 1];
18:               b28 = B[k + 7][j + 1];
19:               tmp1+ = a1*b11+a2*b12+a3*b13+a4*b14+a5*b15+a6*b16+a7*b17+a8*b18
20:               tmp2+ = a1*b21+a2*b22+a3*b23+a4*b24+a5*b25+a6*b26+a7*b27+a8*b28
21:               k+=8
22:             else
23:               tmp1+ = A[i][k] * B[k][j]
24:               tmp2+ = A[i][k] * B[k][j + 1]
25:               k++
26:           C[i][j] = tmp1
27:           C[i][j + 1] = tmp2
28:           j+=2
29:         else
30:           // following code is same as lines 7–19 of Figure 12
```

Figure 16: Resulting program for unrolling the j- and k-loops and removing redundant load statements.

Table 6: Result of execution time and cache misses obtained by Tiling, MDGLIA-j-Unroll, and LCM-j-Unroll

	A.Tiling	B.MDGLIA-j-Unroll	C.LCM-j-Unroll	(A-C)/A	(B-C)/B
time	0.265	0.159	0.203	23.6%	-27.7%
LLC miss	2,083,135	2,338,954	4,079,427	-95.8%	-74.4%

Table 7: Results of execution time and cache misses obtained by each optimization technique with hardware prefetch

	A.Tiling	B.MDGLIA-j-Unroll	C.MDGLIA-j-7-Unroll	(A-B)/A	(A-C)/A
time	0.255	0.157	0.253	38.5%	0.8%
LLC miss	2,006,466	2,091,877	2,028,946	-4.3%	-1.1%

Table 8: Number of spills occurring for Tiling, MDGLIA-j-Unroll, and MDGLIA-j-7-Unroll

	A.Tiling	B.MDGLIA-j-Unroll	C.MDGLIA-j-7-Unroll	B/A	C/A
spills	3	15	69	5.0	23

5.1.5. Discussions

In this evaluation, we used a program that multiplied two matrices, A and B, with sizes 500×500 on a Xeon computer with large cache memories (the associativity of L3 is 20).

In Section 5.1.1, we showed that unrolling the innermost (k-) loop gives us a real situation in which accesses to array A alternate with accesses to array B. Comparing the aggregation with loop tiling, there is no difference between them in the view of LLC misses.

In Section 5.1.2, we showed that unroll-and-jam reveals not only redundant array references, but also references to the higher dimension of array B alternating with references to the lower dimension of the array. Although removing the redundant array references remarkably improves execution efficiency, it increases the number of spills and LLC misses.

In order to perform a simple check of the effectiveness of our aggregation considering array dimension, we compared MDGLIA with LCM, and obtained results that showed that the aggregation significantly reduces the number of LLC misses in Section 5.1.3. This is because matrix multiplication has three loops: i-, j-, and k-loop, and unroll-and-jam to the j-loop gives MDGLIA a chance to aggregate the data for $B[k][j]$ and $B[k][j+1]$. In this program, we used 500×500 matrices; therefore, it is difficult to store a datum of $B[k][j+1]$ in cache memory before use without our aggregation because the associativity is 20.

These results show that removing redundancy and aggregating array references improve the execution efficiency of objective codes. However, removing redundancy typically increases the number of spills, leading to reduced gain in decreasing the cache misses by our aggregation. Therefore, developing a new register allocation that performs spills while considering cache misses may provide powerful support for MDGLIA. In addition, as shown in Section 5.1.4, using MDGLIA with hardware prefetch is good for decreasing cache misses. In the view of execution efficiency, comparing MDGLIA with hardware prefetch with loop tiling, MDGLIA is better than loop tiling. However, in the view of cache misses, MDGLIA with unrolling the j-loop seven times increased the LLC misses about 1.1% although MDGLIA increased the number of spills by a factor of 23 over loop tiling. This result implies that it is able to reduce the LLC miss of MDGLIA to less than that of loop tiling if the new register allocation has been developed.

5.2. Examining the Effect of Speculative Code Motion

In this evaluation, we used the SPEC CPU 2000 benchmark to examine how our speculative code motion contributes to reducing cache misses. We used two CFP2000 programs (equake and art) and three CINT2000 programs (mcf, gzip, and twolf) in the SPEC benchmarks on an x86 machine with an Intel Core2Duo U9600 1.6GHz CPU and CentOS. The cache memory parameters for the x86 machine are shown in Table 9. We kept the default hardware prefetch status because it was not possible to change the status in the BIOS of the computer.

Table 9: Cache memories parameters

Parameters	L1D	L2
Total size (KB)	32	3,072
Line size (bytes)	64	64
Number of cache line	512	49,152
Associativity	8	12

To emphasize the benefits of our algorithm, we compared MDGLIA with the following two optimizations.

LCM-MEM, which only removes redundant array references based on LCM.

GLIA, which aggregates references to the same array without considering their dimensions.

To determine the level of reduction in cache misses occasioned by GLIA and MDGLIA, we measured the following two hardware counters.

DCache_Repl: the number of replacements of L1 data cache.

L2_Lines_Out: the number of removals from the L2 cache.

We simply call these numbers L1D cache miss and L2 cache miss, respectively.

The result of these cache misses for comparison among the three optimizations is shown in Figure 17, and the results for the number of L1D cache misses and L2 cache misses are shown, respectively, in Tables 10 and 11.

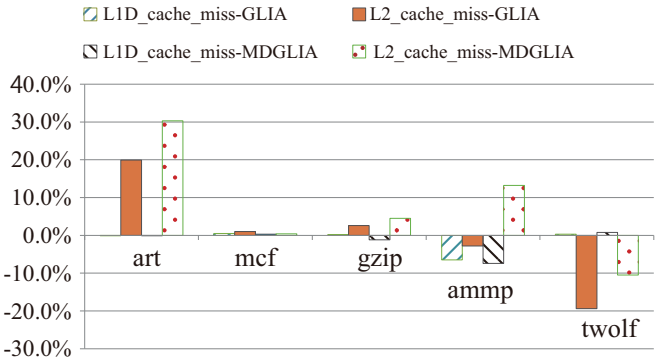

Figure 17: Ratio by which L1D and L2 cache misses are reduced for GLIA and MDGLIA to the cache misses for LCM-MEM.

Comparing GLIA and MDGLIA with LCM-MEM, the number of L1D cache misses is the same degree as LCM-MEM for four programs (art, mcf, gzip, and twolf); however, for one program (ammp) the number increased about 7%. Comparing GLIA with LCM-MEM, the number of L2 cache misses decreased for three programs (art, mcf, and gzip). In particular, the cache miss significantly decreased about 19.9% in art; however, the number of misses increased for two programs (ammp and twolf). Comparing MDGLIA with LCM-MEM, L2 cache miss decreased for four programs (art, mcf, gzip, and ammp). In particular, the cache miss significantly decreased about 30.3% in art; however, the number of misses increased for twolf. Comparing MDGLIA with GLIA, L2 cache miss decreased for four programs (art, gzip, ammp, and twolf). In particular, the cache miss significantly decreased about 15.6% in ammp. In ammp, the algorithms were compared with LCM-MEM, although GLIA increased the number of L2 cache misses in ammp, MDGLIA reduced it. This is because MDGLIA can aggregate more array references than GLIA.

In addition, the execution times of objective codes resulting from each optimization are shown in Figure 18 and Table 12.

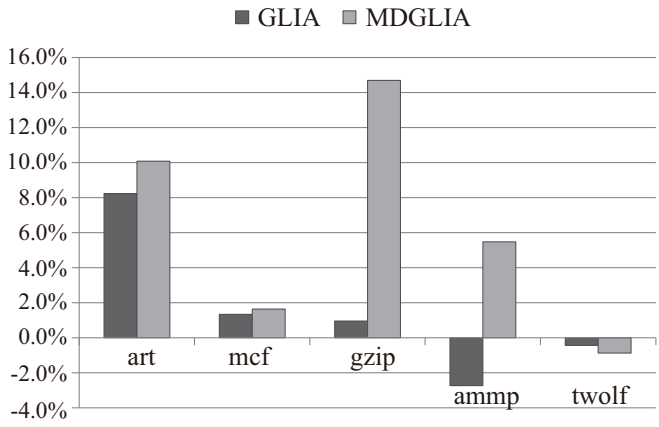

Figure 18: Execution time of objective codes (second).

The results are similar to the result for L2 cache misses. Comparing GLIA with LCM-MEM, the execution time improved for three programs (art, mcf, and gzip), and worsened for two programs (ammp and twolf). Comparing MDGLIA with LCM-MEM, the execution time improved for four programs (art, mcf, gzip, and ammp), and worsened for twolf. Interestingly, in twolf, GLIA and MDGLIA increased the number of L2 cache misses about 10% and 20%, but the execution time worsened about only 0.4% and 0.9%, respectively. In addition, in mcf, comparing MDGLIA with LCM-MEM, the number of L2 cache misses decreased about only 0.4%, but the execution time improved about 1.6%. Comparing MDGLIA with GLIA, the number of L2 cache misses for MDGLIA is greater than that of GLIA, but the execution time of MDGLIA better than GLIA. This is because aggregation moves array references to points that are closer to the start node than the original points; in other words, the range of the hide penalty of cache miss can be lengthened.

In some programs, GLIA and MDGLIA increased cache misses. The increase is considered to be as a result of the following: 1) the number of spills of the temporary variables increased, and 2) speculative code motion lengthened execu-

Table 10: Number of L1D cache misses

Program	A.LCM-MEM	B.GLIA	C.MDGLIA	(A-B)/A	(A-C)/A	(B-C)/B
art	11,439,104,576	11,453,555,362	11,444,685,513	-0.1%	-0.1%	0.1%
mcf	7,567,419,160	7,531,488,486	7,544,975,464	0.5%	0.3%	-0.2%
gzip	7,725,469,653	7,707,389,218	7,818,871,317	0.2%	-1.2%	-1.4%
ammp	21,792,733,488	23,216,563,772	23,413,676,395	-6.5%	-7.4%	-0.8%
twolf	9,056,125,547	9,029,509,531	8,986,903,509	0.3%	0.8%	0.5%

Table 11: Number of L2 cache misses

Program	A.LCM-MEM	B.GLIA	C.MDGLIA	(A-B)/A	(A-C)/A	(B-C)/B
art	366,823,834	293,702,375	255,801,603	19.9%	30.3%	12.9%
mcf	727,422,092	720,133,268	724,349,339	1.0%	0.4%	-0.6%
gzip	25,856,048	25,177,224	24,706,624	2.6%	4.5%	1.9%
ammp	408,120,338	419,706,166	354,268,633	-2.8%	13.2%	15.6%
twolf	1,518,718	1,812,932	1,678,713	-19.4%	-10.5%	7.4%

Table 12: Execution time of objective codes (second)

Program	A.LCM-MEM	B.GLIA	C.MDGLIA	(A-B)/A	(A-C)/A	(B-C)/B
art	97.2	89.2	87.4	8.2%	10.1%	2.0%
mcf	97.6	96.3	96.0	1.3%	1.6%	0.3%
gzip	211	209	209	0.9%	0.9%	0.0%
ammp	475	488	449	-2.7%	5.5%	8.0%
twolf	228	229	230	-0.4%	-0.9%	-0.4%

tion paths without reducing the number of cache misses, as mentioned in Section 4.2.1.

We confirm how the problems increase the number of cache misses below.

5.2.1. Impact of Spill

The number of spills that occur for applications of the three optimizations is shown in Table 13. Comparing GLIA and MDGLIA with LCM-MEM, they caused more spills for four programs (mcf, gzip, ammp, and twolf). In particular, the number of spills increased about 20% in twolf. However, in spite of the increase in the number of spills, the L1D cache misses were held at the same degree with LCM-MEM in mcf and gzip. These additional results show that the increase in the spills does not always equate to an increase in the number of cache misses; in particular, the size of a program is large, such as SPEC CPU benchmark. In fact, in ammp, application of MDGLIA reduced L2 cache misses but increased the number of spills. Note here that, in the previous experiment, matrix multiplication experiment, spills may insert load/store statements in loops because our algorithm increases register pressures in loop. In contrast, in this experiment, SPEC CPU 2000, we applied our algorithm the overall program, including outside of loops. That is, Table 13 includes the number of spills occurred in outside of loops. Conversely, in twolf, the number of spills and cache misses increased. We believe that the load/store statements for spills were inserted between references that were continuously moved by MDGLIA.

5.2.2. Speculative Code Motion vs. Unspeculative Code Motion

In order to determine the number of cache misses that the speculative code motion decreased, we implemented the following two optimizations that do not speculatively aggregate array references:

Table 13: Number of spills

Program	A.LCM-MEM	B.GLIA	C.MDGLIA	(A-B)/A	(A-C)/A	(B-C)/B
art	28	28	28	0.0 %	0.0 %	0.0 %
mcf	61	71	71	-16.4 %	-16.4 %	0.0 %
gzip	107	114	114	-6.5 %	-6.5 %	0.0 %
ammp	317	334	335	-5.4 %	-5.7 %	-0.3 %
twolf	804	962	979	-19.7 %	-21.8 %	-1.8 %

USGLIA, which aggregates array references with the keep-order condition by using *UpSafe*, defined in equation (1) rather than equation (2).

USMDGLIA, which aggregates array references with the keep-order and keep-dimension conditions by using *UpSafe* defined in equation (1) rather than equation (2).

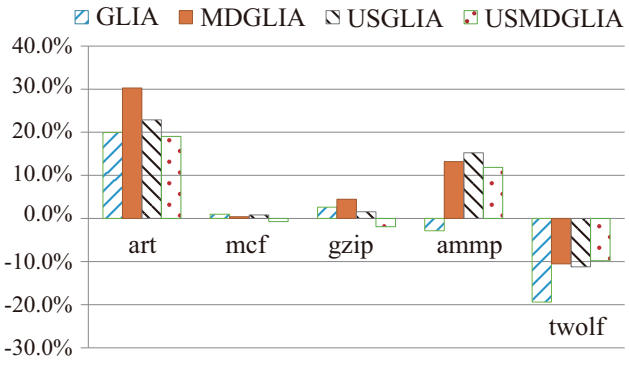

Figure 19: Ratio of reduction in L2 cache misses for GLIA, MDGLIA, USGLIA, and USMDGLIA to the cache miss for LCM-MEM.

The resulting L2 cache misses after applying USGLIA and USMDGLIA are shown in Figure 19 and Table 14. Comparing GLIA with USGLIA, GLIA increased the number of cache misses for three programs (art, ammp, and twolf). In contrast, comparing MDGLIA with USGLIA, MDGLIA reduced the number of cache misses for two programs (art and gzip). Comparing MDGLIA with USMDGLIA, MDGLIA reduced the number of cache misses for four programs (art, mcf, gzip, and ammp). Thus, MDGLIA was better than these unspeculative versions in some programs. This is because speculative code motion can move array references more than unspeculative ones. Table 15 shows the number of stopped delaying by checking keep-order or keep-dimension conditions in the speculative and unspeculative code motions. As shown by the table, the speculative code motion could move more array references than unspeculative one.

Considering L2 cache misses when GLIA, MDGLIA, USGLIA, and USMDGLIA were applied to twolf, they increased L2 cache misses in the program. Although the increase for GLIA was about 20%, the increase for the others increasing was about 10%. Furthermore, comparing USGLIA and USMDGLIA, the results of L2 cache misses are held at the same degree between the two algorithms. This result indicates that speculative insertion under keep-order condition increases the number of cache misses for twolf. On the other hand, speculative insertion under keep-dimension condition decreases the cache misses. That is, the speculative aggregation considering keep-dimension condition reduced the number of L2 cache misses.

5.2.3. Discussion

These results show that speculative code motion is useful for reducing cache misses, compared with simply removing redundant array references. Because the speculative code motion can move array references further than unspeculative one, but unspeculative code motion is sometimes better than speculative ones. We believe that extension of MDGLIA for analyzing whether the aggregated array references will get a cache hit using profile information has the potential to significantly reduce cache misses.

Table 14: Number of L2 cache misses of USGLIA and USMDGLIA, and the ratio of their cache misses to the cache miss for GLIA and MDGLIA

Program	USGLIA	vs. GLIA	vs. MDGLIA	USMDGLIA	vs. MDGLIA
art	282,855,332	3.7 %	-10.6 %	297,012,498	-16.1 %
mcf	721,268,586	-0.2 %	0.4 %	732,708,340	-1.2 %
gzip	25,449,483	-1.1 %	-3.0 %	26,351,879	-6.7 %
ammp	345,935,455	17.6 %	2.4 %	359,701,112	-1.5 %
twolf	1,689,141	6.8 %	-0.6 %	1,667,434	0.7 %

Table 15: Number of aggregated array references by working *keepOrder* and *keepDimension* in speculative/unspeculative code motion

	Speculative		Unspeculative	
Program	*keepOrder*	*keepDimension*	*keepOrder*	*keepDimension*
art	438	22	145	22
mcf	750	298	104	71
gzip	829	134	182	77
ammp	8,043	2,015	719	512
twolf	30,399	7,313	8,855	811

As stated in Section 3.1, MDGLIA assumes that array data is placed in row-major order. However, recently, Hilbert space-filling curve was used rather than row-major order. This trend means that it is effective for aggregating array references to use profile information for switching data-layouts. In addition, the usage of profile information may enhance aggregation by estimating speculation penalties and heavy uses of pointers. Because the current algorithm of MDGLIA does not handle pointer operations sophisticatedly, the extension with the-state-of-art alias or points-to analysis techniques [12, 25] may improve some practical programs such as SPEC benchmarks.

6. Conclusions

In this paper, we proposed a new global code motion algorithm that aggregates array references with the same indexes for the same array in order to reduce the number of cache misses. Our algorithm not only reduces the number of cache misses but also suppresses spills by delaying the array references without changing their access order.

To demonstrate the effectiveness of our algorithm, we applied it to matrix multiplication and SPEC CPU 2000 benchmark programs. Consequently, we showed that our aggregation significantly reduces the number of cache misses. The results of evaluation of matrix multiplication indicate that removing redundant array references sometimes reduces the gain of the aggregation because of increasing spills; therefore, developing a new register allocation is one of the most important future works. From the evaluation of SPEC CPU 2000, we found that speculative code motion can reduce the number of cache misses, but sometimes increases it. We believe that extending MDGLIA to use profile information in order to analyze the aggregated array references has the potential to further reduce cache misses.

Acknowledgements

We would like to thank the reviewers for their valuable and insightful comments and suggestions that helped improve the paper. This research is partially supported by Scientific Research (25330089).

References

[1] Aho, A.V., Sethi, R., Ullman, J.D., 1986. Compilers: principles, techniques, and tools. Addison-Wesley

Longman Publishing Co., Inc., Boston, MA, USA.

[2] Appel, A.W., 1997. Modern Compiler Implementation in ML: Basic Techniques. Cambridge University Press, New York, NY, USA.

[3] Bodik, R., Gupta, R., Soffa, M.L., 1998. Complete removal of redundant expressions, in: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, ACM, New York, NY, USA. pp. 1–14. URL: http://doi.acm.org/10.1145/277650.277653, doi:10.1145/277650.277653.

[4] Calder, B., Krintz, C., John, S., Austin, T., 1998. Cache-conscious data placement, in: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems, ACM, New York, NY, USA. pp. 139–149. URL: http://doi.acm.org/10.1145/291069.291036, doi:10.1145/291069.291036.

[5] Carr, S., Kennedy, K., 1994. Scalar replacement in the presence of conditional control flow. Softw. Pract. Exper. 24, 51–77. URL: http://dx.doi.org/10.1002/spe.4380240104, doi:10.1002/spe.4380240104.

[6] Chaitin, G.J., 1982. Register allocation & spilling via graph coloring, in: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, ACM, New York, NY, USA. pp. 98–105. URL: http://doi.acm.org/10.1145/800230.806984, doi:10.1145/800230.806984.

[7] Dhamdhere, D.M., 1988. A fast algorithm for code movement optimisation. SIGPLAN Not. 23, 172–180. URL: http://doi.acm.org/10.1145/51607.51621, doi:10.1145/51607.51621.

[8] Dhamdhere, D.M., Patil, H., 1993. An elimination algorithm for bidirectional data flow problems using edge placement. ACM Trans. Program. Lang. Syst. 15, 312–336. URL: http://doi.acm.org/10.1145/169701.169684, doi:10.1145/169701.169684.

[9] Geng, Y., Huang, X., Yang, G., 2014. Swiftarray: Accelerating queries on multidimensional arrays. Tsinghua Science and Technology 19, 521–530. doi:10.1109/TST.2014.6919829.

[10] Gupta, R., Berson, D.A., Fang, J.Z., 1998. Path profile guided partial redundancy elimination using speculation, in: Proceedings of the 1998 International Conference on Computer Languages, IEEE Computer Society, Washington, DC, USA. pp. 230–. URL: http://dl.acm.org/citation.cfm?id=857172.857261.

[11] Gupta, R., Bodik, R., 1999. Register pressure sensitive redundancy elimination, in: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Springer-Verlag, London, UK, UK. pp. 107–121. URL: http://dl.acm.org/citation.cfm?id=647475.727623.

[12] Hardekopf, B., Lin, C., 2009. Semi-sparse flow-sensitive pointer analysis, in: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA. pp. 226–238. URL: http://doi.acm.org/10.1145/1480881.1480911, doi:10.1145/1480881.1480911.

[13] Ishitobi, Y., Ishihara, T., Yasuura, H., 2010. Code and data placement for embedded processors with scratchpad and cache memories. J. Signal Process. Syst. 60, 211–224. URL: http://dx.doi.org/10.1007/s11265-008-0306-3, doi:10.1007/s11265-008-0306-3.

[14] Kawahito, M., Komatsu, H., Nakatani, T., 2004. Partial redundancy elimination for access expressions by speculative code motion. Softw. Pract. Exper. 34, 1065–1090. URL: http://dx.doi.org/10.1002/spe.604, doi:10.1002/spe.604.

[15] Knoop, J., Ruthing, O., Steffen, B., 1992. Lazy code motion, in: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, ACM, New York, NY, USA. pp. 224–234. URL: http://doi.acm.org/10.1145/143095.143136, doi:10.1145/143095.143136.

[16] Lo, R., Chow, F., Kennedy, R., Liu, S.M., Tu, P., 1998. Register promotion by sparse partial redundancy elimination of loads and stores, in: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, ACM, New York, NY, USA. pp. 26–37. URL: http://doi.acm.org/10.1145/277650.277659, doi:10.1145/277650.277659.

[17] Mahlke, S.A., Chen, W.Y., Bringmann, R.A., Hank, R.E., Hwu, W.M.W., Rau, B.R., Schlansker, M.S., 1993. Sentinel scheduling: A model for compiler-controlled speculative execution. ACM Trans. Comput. Syst. 11, 376–408. URL: http://doi.acm.org/10.1145/161541.159765, doi:10.1145/161541.159765.

[18] Morel, E., Renvoise, C., 1979. Global optimization by suppression of partial redundancies. Commun. ACM 22, 96–103. URL: http://doi.acm.org/10.1145/359060.359069, doi:10.1145/359060.359069.

[19] Odaira, R., Hiraki, K., 2004. Partial value number redundancy elimination. Information Processing Society of Japan Transactions on Programming 45, 59–79. (in Japanese).

[20] Poletto, M., Sarkar, V., 1999. Linear scan register allocation. ACM Trans. Program. Lang. Syst. 21, 895–913. URL: http://doi.acm.org/10.1145/330249.330250,

doi:10.1145/330249.330250.

[21] Sarkar, S., Tullsen, D.M., 2008. Compiler techniques for reducing data cache miss rate on a multithreaded architecture, in: Proceedings of the 3rd international conference on High performance embedded architectures and compilers, Springer-Verlag, Berlin, Heidelberg. pp. 353–368.

[22] Smith, A.J., 1982. Cache memories. ACM Comput. Surv. 14, 473–530. URL: http://doi.acm.org/10.1145/356887.356892, doi:10.1145/356887.356892.

[23] Sumikawa, Y., Takimoto, M., 2013. Effective demand-driven partial redundancy elimination. Information Processing Society of Japan Transactions on Programming 6, 33–44.

[24] Sumikawa, Y., Takimoto, M., 2014. Global load instruction aggregation based on array dimensions, in: Proceedings of the 2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming, IEEE Computer Society, Washington, DC, USA. pp. 123–129. doi:10.1109/PAAP.2014.43.

[25] Zheng, X., Rugina, R., 2008. Demand-driven alias analysis for c, in: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA. pp. 197–208. URL: http://doi.acm.org/10.1145/1328438.1328464, doi:10.1145/1328438.1328464.

Yasunobu Sumikawa received his B.S. degree in Mathematics from Tokyo University of Science in 2010, and his M.S. and Ph.D. degrees in Information Science from Tokyo University of Science in 2012 and 2015, respectively. He is currently an assistant professor at Tokyo University of Science. His research interests lie in compiler and data mining.

Munihiro Takimoto is a professor in the Department of Information Sciences at Tokyo University of Science. His research interests include the design and implementation of programming languages. He received his undergraduate, postgraduate, and doctoral degrees in engineering from Keio University.