Computers and Electrical Engineering 50 (2016) 180–199

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Global load instruction aggregation based on dimensions of arrays☆

Yasunobu Sumikawa*, Munehiro Takimoto

Tokyo University of Science, 2641, Yamazaki, Noda, Chiba 278-8510, Japan

ARTICLE INFO

Article history:
Received 1 December 2014
Revised 28 August 2015
Accepted 31 August 2015
Available online 21 October 2015

Keywords:
Partial redundancy elimination
Cache optimization
Register pressure
Data-flow analysis

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.

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 Fig. 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

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Guest Editor Dr. Y. Sang.

* Corresponding author.
E-mail addresses: yas@cs.is.noda.tus.ac.jp (Y. Sumikawa), mune@cs.is.noda.tus.ac.jp (M. Takimoto).
URL: http://www.cs.is.noda.tus.ac.jp/~yas (Y. Sumikawa)

http://dx.doi.org/10.1016/j.compeleceng.2015.08.020
0045-7906/

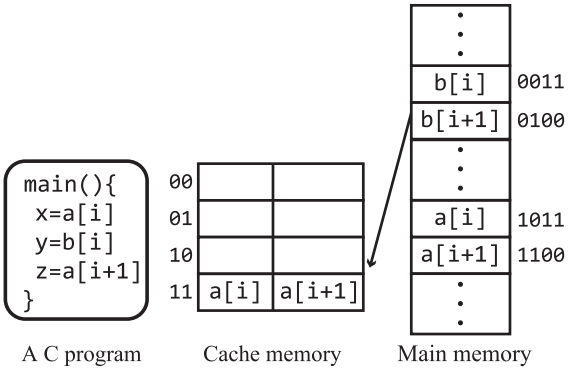

Fig. 1. An example C program that loads data from two kinds of arrays, and causes a cache miss on which our algorithm is focused.

data is transferred from main memory to cache memory, the candidate location in which it is placed is determined by the memory address modulo the number of lines in the cache memory. 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 Fig. 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 Fig. 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 Fig. 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) [18,3,15] 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 Fig. 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, 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

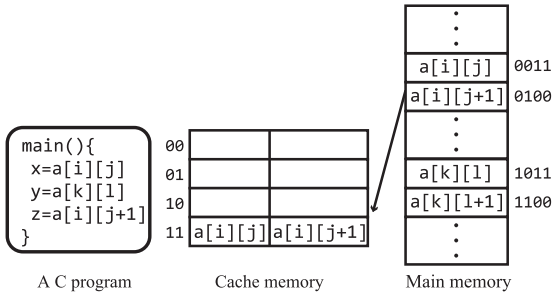

Fig. 2. An example C program that loads data from a multidimensional array, and causes a cache miss on which our algorithm is focused.