Address Correlation: Exceeding the Limits of Locality

Resit Sendag, Peng-fei Chuang, and David J. Lilja
Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota, Minneapolis, MN 55455
E-mail: {rsgt, pengfei, lilja}@ece.umn.edu

***Abstract*[1]—We investigate a program phenomenon, *Address Correlation*, which links addresses that reference the same data. This work shows that different addresses containing the same data can often be correlated at run-time to eliminate a load miss or a partial hit. For ten of the SPEC CPU2000 benchmarks, 57 to 99% of all L1 data cache load misses, and 4 to 85% of all partial hits, can be supplied from a correlated address already found in the cache. Our source code-level analysis shows that semantically equivalent information, duplicated references, and frequent values are the major causes of address correlations. We also show that, on average, 68% of the potential correlated addresses that could supply data on a miss of an address containing the same value can be correlated at run time. These correlated addresses correspond to an average of 62% of all misses in the benchmark programs tested.**

I. Introduction

As processor clock speeds have increased along with microarchitectural innovations, the gap between processor and memory performance has become a greater bottleneck. Mechanisms such as caches have been introduced to close this gap. While a conventional cache system relies heavily on the temporal and spatial locality that programs exhibit, a recently introduced observation, value locality, has proven to be a powerful supplement to the cache system effectiveness. Value locality [1] describes the recurrence of a previously seen value within a storage location. It allows the classical dataflow limit to be exceeded by executing instructions before their operand values are available.

There have been several studies on exploiting different types of value locality, such as store value locality [2] and frequent value locality [3]. *Store value locality* introduced the concept of redundancy in data that was stored to memory. *Frequent value locality* showed that a few values appear very frequently in memory locations and are therefore involved in a large fraction of all memory accesses.

Extending these two complementary studies, we propose a new technique, *Address Correlation*, which is based on the dynamic linking of addresses that store the same value. This run-time correlation can be used to improve the performance of on-chip data caches by forwarding data to the processor on a miss that is already resident in the cache at other correlated addresses. Our results show that there is substantial potential for hiding memory latency by providing the data from a correlated address instead of incurring a full miss penalty. We demonstrate that, in addition to reducing cache misses, address correlation can be effective in servicing partial hits[2] faster.

We also present a detailed source code-level analysis of programs to demonstrate the causes of address correlation. We find that semantically equivalent information, duplicated references, and frequent values are the major causes of the address correlations. Taking advantage of duplicated references has excellent potential to benefit object-oriented programs due to their extensive usage of aggregation classes.

The remainder of this paper is organized as follows. Section II provides the background information and motivation for address correlation. Section III gives the detailed source code-level analysis to help to better understand the sources of address correlation. In Section IV, we classify the address correlations. Section V presents some upper-bound performance results while we conclude in Section VI. A realistic hardware implementation that exploits this program behavior is beyond the scope of this paper, although it is currently under investigation.

II. Address Correlation

In this section, we present the profiling results that motivate the idea of address correlation and describe the basic operation of an address correlation system. We profiled selected MinneSPEC [4] CPU2000 benchmarks (with O3 optimization) to test the potential for a data miss to be found in another address residing in the cache. Our microarchitectural simulator is built on top of the SimpleScalar toolset [5], version 3.0. The processor/memory model used in this study is capable of issuing 8 instructions per cycle using out-of-order execution.

In Figure 1, we show the percentage of all data misses and partial hits whose values can be found in other addresses in the L1 data cache. It can be seen that 57 to 99% of all data cache load misses can be serviced by correlated addresses, and that 4 to 85% of all partial hits can be serviced faster using address correlation.

The potential seen in Figure 1 for supplying data from another address in the L1 data cache on a miss or a partial hit of a requested address suggests that it may be useful to correlate the addresses that reference the same data. An *address correlation system* (ACS) that correlates these

[1] This work was supported in part by National Science Foundation grants EIA-9971666 and CCR-9900605, the IBM Corporation, Compaq's Alpha development group, and the Minnesota Supercomputing Institute.

Manuscript submitted: 27 Mar. 2003. Manuscript accepted: 13 May 2003. Final manuscript received: 23 May 2003.

[2] A partial hit occurs when a request on an address is a hit in the cache, but the data at the address is not ready yet because it is in the process of being read by a previous miss in the same cache block. A partial hit can be as slow as a complete cache miss, depending on how close together the two accesses occur.

addresses at run-time makes it possible to hide the latency for many memory references. Run-time address correlation requires tracking the contents of the cache, the relationships between different locations in the cache, and the memory access history.

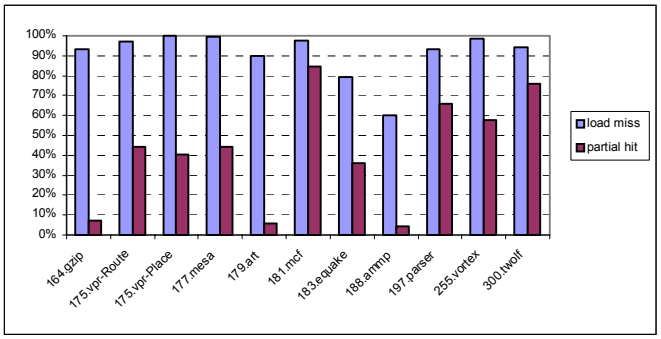


Fig. 1. The percentage of all load misses and partial hits whose values are found in other addresses. The L1 data cache is 32KB and 4-way associative.

In our idealized implementation, the ACS stores the addresses evicted from the cache and their correlated addresses, which remain in the L1 data cache to supply data for the evicted addresses. The ACS provides information for an alternative data source on a cache miss or a partial hit. When a cache line is brought into the L1 data cache, all the addresses in that cache line are linked with the addresses of other cache lines according to their values. An address will never be linked with the addresses in the same cache line since they all will be evicted from the cache at the same time. If a new value is stored to an address, all the links to the updated address are removed and new links are generated for the address based on its new value. In order to study the upper-bound potential of address correlation, we used an infinite table for storing the correlations between addresses at run-time. This configuration allows an unlimited number of correlations since there are no evictions from the correlation table.

Although specific optimizations to efficiently design an ACS are not proposed here, we expect that this first understanding of why address correlation is possible, and investigating the sources of address correlation in application programs, will naturally lead to appropriate uses for this information.

III. INVESTIGATING PROGRAM BEHAVIOR

A better understanding of the sources of address correlation lies in the behavior of the application programs. Thus, in this section, we concentrate on analyzing the program source code in order to understand the relation between the correlated addresses, the load/store instructions that brought them into the cache, and the interaction between these instructions.

A. Correlation between fields of structures

In many instances of a programmer-defined structure, there often are fields that contain the same values. When a program creates a large array of these objects, the fields with the same data in different objects can be correlated. For example, consider a database of students in a high school. The *Student* entity may have attributes such as *state*, *city*, etc., which are likely store the same value for many of the students.

The *181.mcf* benchmark demonstrates this kind of behavior. In Figure 2, the *arc_t* structure contains a field called *ident*. Profiling results show that this *ident* field usually stores the same value in different instances of the *arc_t* type. These different addresses with the same *ident* values then can be correlated. Five percent of the misses found by correlation in *181.mcf* are due to the misses caused by loading *ident*.

```
typedef struct node {
   cost_t potential;
...
} node_t;
```

```
typedef struct arc {
   node_t *tail, *head;
   cost_t cost, org_cost;
   long ident;
...
} arc_t;
```

Fig. 2. The definitions of the *arc_t* structure and the *node_t* structure in *181.mcf*.

Figure 3 demonstrates another example of a useful address correlation in *181.mcf*. The correlation in this case can be found in fields of different structures. The *potential* field of *node_t* and the *cost* field of *arc_t* often store the same value. Therefore, in the function *bea_compute_red_cost(),* the variables *arc->tail->potential, arc->head->potential* and *arc->cost*, are all correlated with each other. We found that 52% of the misses eliminated by address correlation in *181.mcf* are due to the misses caused by loading these fields.

```
cost_t bea_compute_red_cost( arc_t *arc ) {
  return arc->cost - arc->tail->potential + arc->head->potential;
}
```

(a) function *compute_red_cost()*

```
arc_t *primal_bea_mpp( long m,  arc_t *arcs, arc_t *stop_arcs,
                       cost_t *red_cost_of_bea )  {….
  for( i = 2, next = 0; i <= B && i <= basket_size; i++ ) {
    .....    red_cost = bea_compute_red_cost( arc );.....  }

  for( ; arc < stop_arcs; arc += nr_group ) {
    .....    red_cost = bea_compute_red_cost( arc );.....}
…...}
```

(b) A function calling *compute_red_cost(). primal_bea_mpp()* is a frequently used function and calls *compute_red_cost()* in two of its loops.

Fig. 3. Example code segments from *181.mcf*

The actual value stored in these fields may vary during the program's execution. However, during a given execution phase, the same value usually occupies most of the instances of the structures. For example, the value of *ident* contains only a limited number of defined constants, i.e., BASIC, FIXED, AT_LOWER, and AT_UPPER.

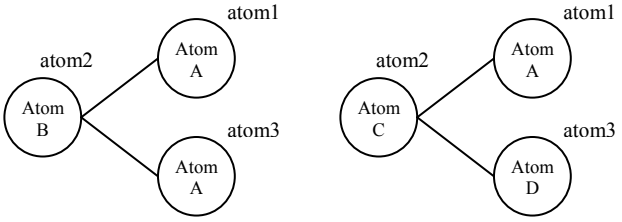
B. Correlation between references to instances

Consider a data structure that encapsulates instances of another basic structure. If an instance of the basic structure is duplicated in several instances of the encapsulating structure, we may find correlations between those references to the instance of the basic structure among all the instances of the

structure using it.

An example of this behavior can be seen in Figure 4 for *188.ammp*, which is a computational chemistry application. This program uses structures to store molecular information. Since different molecules can contain the same atoms, a single atom structure may be referenced by several different molecule structures. The addresses that reference the same atom can be correlated when instances of molecules are formed. While accessing atom references, a miss on an address that references a specific atom may be eliminated by using the references to the same atom by other molecular structures that already reside in the cache.

This type of correlation is especially useful for object-oriented design. The idea of encapsulating structures can be extended to aggregation classes (the structure of a class whose encapsulated data includes references to instances of other classes), which play an essential role in class hierarchy design. The use of aggregation classes in object-oriented programming is likely to produce substantial address correlation.



(a) Two different molecules using some of the same component atoms.

```
typedef struct{
    ATOM *atom1,
      *atom2,*atom3;
    ......
} ANGLE;
```

(b) The user-defined type in *188.ammp* for storing the angle information

Fig. 4. In *188.ammp*, the ANGLE structure, which consists of three atoms, is part of a molecule. Providing we have two different ANGLEs, as depicted in (a), to be stored in two instances of the ANGLE structure, say angle1 and angle2, we would find the values of references to angle1.atom1, angle1.atom3 and angle2.atom1 to be the same.

```
 1  ATOM *a_m_serial(serial)
 2  int serial;
    {
 3      static ATOM *ap = NULL;
 4      static ATOM *lastmatched = NULL;
 5      int i , n, a_number();
    ......
 6      for( i=0; i< n; i++ ) {
 7         if(ap-> serial == serial)
              {lastmatched = ap; return ap;}
 8         if(ap == ap->next) ap = first ;
 9         else ap = ap->next;
        }
10      return NULL;
    }
```

Fig. 5. An example from *188.ammp* that traverses the ATOM linked list.

Address correlation is sometimes useful when accessing recursive data structures such as linked list traversals. Figure 5 shows a function called many times in *188.ammp*. Given the serial number of an atom, this function returns the reference to the atom structure. We found that 43% of the misses eliminated by address correlation in *188.ammp* are caused by the loading of *ap->next* at lines 8 and 9 (a single load instruction is generated by the compiler for these two statements). The misses caused by this load could be supplied 64% of the time by another reference in the linked list[3], 11% are supplied by the variable, *lastmatched*[4], and the remaining 25% are supplied by references from other structures embedding atoms.

C. Correlations of frequent values

Frequent values are another major source of correlations. Intuitively, the more copies of a value that exist in the cache, the more often useful correlations can be created. For example, zero is extensively used for variable initialization, for constants such as NULL or FALSE, to fill sparse matrices, and as the starting value of enumeration types. We expect to see a large portion of the correlated address to come from frequent values such as zero.

IV. Classifying the Address Correlations

An address can be brought into the cache in two ways. One way is through explicit memory accesses where the address was the target of some load or store instruction. We call this type of address a *requested* address. The other way is that an address is put into the cache along with some requested address because the two addresses are located in the same cache line. We refer to these addresses as *non-requested* addresses. In this study, both types of addresses can be correlated. Based on the reason an address is in the cache, we can classify address correlations into two categories: 1) correlations between two requested addresses, and 2) correlations involving at least one non-requested address.

The correlations containing at least one non-requested address are more likely to occur due to frequent or trivial values. In Figure 6, a hit in the ACS is categorized according to the types of the two correlated addresses and the value in these addresses. If a missed address can be supplied both by requested addresses and non-requested addresses, a requested address is employed. Zero is one of the frequent values appearing in the SPEC CPU2000 benchmarks. While it might not be the most frequently accessed or occurring value for all benchmarks, it provides good insight into the effect that frequent values may have on address correlation.

Figure 6 shows that, on average, 12% of the misses serviced by address correlation are due to the value zero, when the correlated addresses are both requested addresses. On the other hand, 23% of the misses serviced by address correlation are due to the correlation of non-requested addresses that contain zero. Further investigation shows that, for six of the

[3] The ATOM linked list in *188.ammp* has an unusual design in that the last node references back to itself instead of NULL. This allows the next field of the last node in the linked list to be correlated with the next field of the second-to-the-last node.

[4] *lastmatched* is a static variable that stores the reference to the ATOM node that was the matched target in the previous call to this searching function.

benchmarks, 60% of the misses eliminated by address correlation are due to only four distinct values.

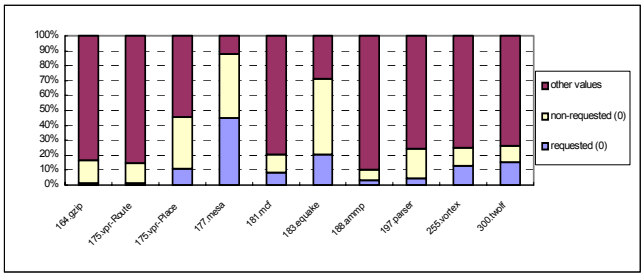


Fig. 6. To determine the importance of frequent values in address correlation, ACS hits are categorized into "0" and "others". The value 0 is further divided into correlations between two requested addresses (requested), and correlations involving at least one non-requested address (non-requested).

V. Upper-bound Potential of Address Correlation

In this section, we present the upper-bound performance improvement made possible by address correlation. While all of the potential addresses that store the same data can be candidates to supply the data on a miss or a partial hit, all of them might not be correlated to the requested address. Our intent is to see if all the misses that can be found in other addresses inside the cache can be correlated at run-time, and thus can supply the data. This will give us the upper-bound potential that an address correlation mechanism can offer for exceeding the limits of locality.

In Figure 7, we show the normalized L1 data cache miss counts for different cache sizes and the effect of address correlation on reducing the number of misses. The original superscalar processor with an 8KB, 4-way associative L1 data cache is used as the base for these comparisons. We see that address correlation can eliminate most of the misses in the L1 data cache. That is, the data requested is usually already in the cache at other addresses.

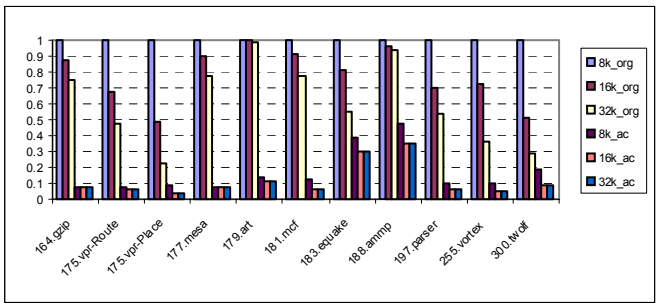


Fig. 7. The normalized cache miss counts for varying L1 data cache sizes with and without address correlation (ac). The original (org) superscalar processor with an 8KB, 4-way associative L1 data cache is used as the base for these comparisons.

While these results show the potential benefits of the address correlation mechanism, the important question is how successfully it can correlate addresses to supply the requested data from an alternative address. Figure 8 shows that, on average, 68% of the potential addresses that can supply the data on a miss can be correlated at run-time by an ACS mechanism and thus can supply the data on a miss of an address containing the same value. While Figure 1 showed that an average of 91% of the load misses can be found in other addresses residing in the cache, we see in Figure 8 that fewer misses can be eliminated at run-time by an ACS. Nevertheless, the reduction still is significant, ranging from 23 to 99%, with an average of 62%.

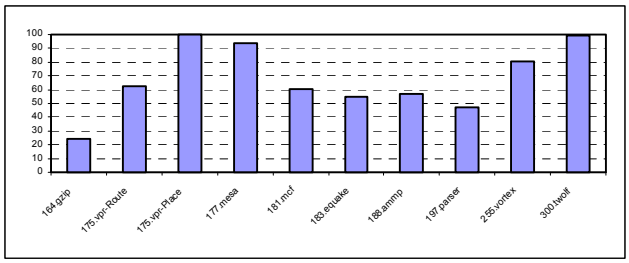


Fig. 8. The percentage of L1 data cache misses eliminated at run-time by an ACS. The L1 data cache is 32KB with 4-way associativity.

VI. Conclusion

This paper has demonstrated a new approach for exploiting value locality. Based on the concepts of store value locality [2] and frequent value locality [3], we proposed a new technique, *Address Correlation*, to link different addresses that contain the same data. We showed that supplying the requested data from different addresses that contain the same data value can substantially reduce the data cache misses and can also service partial hits faster. Our detailed source code-level analysis of programs shows that semantically equivalent information, duplicated references, and frequent values are the major causes of the address correlations. Taking advantage of duplicated references has further potential to benefit objected-oriented design with its extensive usage of aggregation classes.

The next step of this study is to develop a feasible implementation of the ACS. According to our profiling results, a useful correlation can usually be found in cache lines that are physically close to each other. Furthermore, the number of addresses that can be usefully correlated is usually bounded. Our preliminary experiments show that an ACS with 1-2 correlations for a value can usually provide comparable performance results to that of the upper bound study given in this paper. We anticipate that the number of addresses that have correlations also can be limited with an effective replacement policy. In addition, with an efficient algorithm for identifying these addresses, we can create the correlation with low hardware overhead and low searching latency.

References

[1] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, 1996.

[2] K. M. Lepak, G. B. Bell, and M. H. Lipasti, "Silent Stores and Store Value Locality," *IEEE Transactions on Computers*, Vol. 50, No. 11, 2001.

[3] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-centric Data Cache Design," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 150-159, 2000.

[4] AJ KleinOsowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, Volume 1, May 2002.

[5] D.C. Burger, T.M. Austin, and S. Bennett, *Evaluating Future Microprocessors: The SimpleScalar Tool Set*, Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.