Reuse Distance Analysis

Chen Ding Yutao Zhong

Computer Science Department
University of Rochester
Rochester, New York
{cding,ytzhong}@cs.rochester.edu

February 2001

Technical Report UR-CS-TR-741

Abstract

Cache is one of the most widely used components in today's computing systems. Its performance is heavily depended on the locality in programs. Till now, the analysis of program locality relies on expensive cache simulation. As machine cache becomes increasingly complex and adaptive, more efficient and accurate methods are needed to find the best cache configuration for each program or even each part of the program.

In this report, we measure program locality directly by the distance between the reuses of its data. Data reuse is an inherent program property and does not depend on any cache parameters. Therefore, it allows quantitative measurement of program locality that is not tied to any particular machine. To measure reuse distance, we describe a new method consisting of two components. The first performs fast analysis for full applications accessing large data sets, and the second ascribes the simulation result to source-level data structures at fine granularity. With this tool, we analyze data reuse behavior in a set of benchmark applications and present main findings about their program locality.

1 Introduction

One of the most important concepts in today's computing systems is cache, which improves the access speed to a large, far away memory by storing frequently used data in a smaller, local buffer. Cache is used on all current microprocessors between CPU and main memory as well as other parts of a computing system such as networking system, file system, and databases. System performance heavily depends on the performance of cache.

The effectiveness of cache hinges on a term called program locality. A good locality means good utilization of cache. Unfortunately, this is a circular definition: cache performance is determined by locality while locality is measured by cache performance. Till now, most cache analysis studies miss rate on a fixed cache rather than analyzing the overall locality of a program.

Some studies indirectly measure program locality by simulating its execution on a set of probable cache configurations. Although time-consuming, it is not accurate since not all cache sizes are examined. The best cache size could be one that is between the two simulated sizes. The problems of efficiency and accuracy become especially acute with the emergence of adaptive architecture, where cache size and other parameters can be dynamically adjusted across applications or even inside the same application. Adding to this problem is the use of multi-level cache, which exponentially increases the search space for best cache combination. Therefore, exhaustive simulation is not practical for determining the best cache configuration for every segment of every program running on every set of data input.

To provide a measure for program locality, we propose a new model based on *reuse distance*, which is the frequency of data reuse in programs. Measuring data reuse in programs, as opposed to blocks reuse in cache, has three important advantages. First, data reuse is independent of any aspect of cache configuration because it is purely a property of programs. Therefore, it allows quantitative comparison between programs that is not tied to any particular machine. Second, data reuse is a main determinant in cache performance because all cache reuse comes from reuse of the same or adjacent data, regardless of the organization of cache. Therefore, reuse distance separates program-specific factors from machine-specific factors. This leads to the third benefit—the division allows us to analyze and optimize inherent program data behavior separately from their interaction with a particular cache design.

In this paper, we describe and evaluate a new method for measuring reuse distance. We aim at two goals. The first is fast analysis: we want to measure large programs with realistic data inputs. We will use a method originally designed by Bennett and Kruskal in 1975 [1], instead of the commonly used stack algorithm by Mattson et al [5]. We refer to the first algorithm as *counting* algorithm in the rest of this report. Our second goal is accurate presentation: we want to describe program locality in terms of program structure rather than program trace. We will use a program instrumentor to attribute analysis result to source-level data structures. The information of source-level data structures significantly improves the efficiency of the counting algorithm because of the reduction in data name space. With this new method, we will analyze a set of commonly used benchmark programs. We will present the main findings about their program locality, as well as the efficiency of our analysis method.

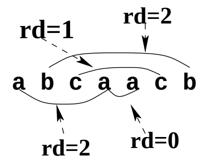
The rest of this paper is organized as follows. Section 2 defines reuse distance and the measurement method including the simulator, the program instrumentor, and their runtime connections. Section 3 evaluates our method on a number of commonly used benchmark programs. Section 4 reviews related work and finally, Section 5 concludes.

2 Reuse Distance and Its Measurement

2.1 Reuse Distance

From the view of memory and cache, a sequential execution of a program is a sequence of data access. Given such a sequence, we define *reuse distance* as the number of distinctive data elements accessed between two consecutive uses of the same element. If the notion of time is not defined by the number of clock cycles but by the number of memory references, reuse distance is equivalent to the temporal reuse of data. Unlike the traditional notion of temporal reuse, reuse distance is precise defined and machine independent. Figure 1 shows an example sequence and the distance for each pair of data reuses. The distance between the reuses of b is two because two distinct data elements, a and c, are accessed in between.

Reuse distance largely determines cache performance because all cache reuse comes from data or data



(a) Example sequence and its reuse distances

Figure 1: Reuse Distance

block reuse. Of course, the exact cache performance also depends on cache configurations such as cache size, associativity and replacement policy. However, for fully associative LRU cache, reuse distance can accurately measure the number of cache hits or misses: a data reference misses in cache if and only if the reuse distance is larger than or equal to the cache size. Such a miss is often called a capacity miss [4].

For cache with non-unit size cache blocks, the reuse of the same cache block can be triggered by references to neighboring data elements. The reuse is called *cache spatial reuse*. The amount of cache spatial reuse is determined by the memory layout of the program and the size of cache blocks. Once these factors are known, reuse distance can also measure spatial reuse by modeling the distance of data blocks rather than data elements. We do not study spatial reuse in this report and the remaining sections will only consider temporal reuse.

To measure the overall locality, we use a histogram of the reuse distance of all memory references. The shorter reuse distances a program has, the better its temporal reuse is. In extreme cases, a program whose reuse distances are far below the cache size will likely to have a perfect cache performance, while a program whose reuse distances are far greater than cache size may not utilize cache at all. For fully associative LRU cache, the histogram of reuse distances determines the miss rate: those and only those references whose reuse distance is larger than cache size are cache misses. Throughout this report, we will use the histogram of reuse distances to study program locality. Next, we describe the collection of the histogram.

2.2 Measuring Reuse Distance

This section shows how to measure all reuse distances given a trace of data access. The next two sections will describe how to collect the trace from programs and how to combine the trace collection and the simulation into one pass. The counting method we describe here is due to Bennett and Kruskal in 1975 [1].

Given a data access trace, we define the *access time* of a memory reference as its index position in the trace, counting from the first reference of the program. Given the next memory reference and its previous access time, the algorithm in Figure 2 finds the reuse distance by counting the number of distinct data elements in between, and then updating the trace, the histogram and the last-access time of the element. The counting is the key step of the algorithm, which we describe next. Section 2.4 will explain how to store and retrieve the last access time of data elements.

```
Algorithm MeasureReuseDistance(elem, lastAccess)
  // elem: the data being accessed
  // lastAccess: the time of last access to elem, initialized to -infinity

  trace(1...t): trace of past memory references, initially empty

  let t be the next empty cycle in trace
  trace(t) = elem
  t1 = lastAccess
  reuseDis = CountDistinctElems(t1,t,trace)
  lastAccess = t
  record reuseDis in the histogram
End MeasureReuseDistance
```

Figure 2: Algorithm for measuring reuse distance

A straightforward method of counting is to visit every cell of the trace and count the first appearance of each element. Three techniques can make the process faster. First, we keep only the last access of each element in trace and empty the cells that contain its previous uses. For example, if element y is accessed multiple times before x, only the last access is kept in the trace. The reuse distance is accurate because it starts from the current access, say x, and counts y if and only if it is accessed after the previous use of x. Thus, the reuse distance of x is the number of non-empty trace cells between two references of x. In fact, the trace can be simplified into a bit vector. The second improvement is that we block the trace into segments of a constant size B in order to quickly count non-empty cells. Each block maintains the total number of its non-empty cells. When counting the whole block, we use the block count directly instead of visiting all block elements. Finally, we organize block counters as a m-ary tree (or B-tree). The summation and update of counters would access only the logarithmic number of the counted elements on the trace.

The counting method with the above three improvements is shown Figure 3. Given a segment of the trace, the algorithm first counts elements in partial blocks and then uses block count to calculate reuse distance. The second part of the algorithm maintains the data structures by emptying the cell of the second last access and by adjusting the block counts. The algorithm is adapted from our actual implementation in C, following its conventions of integer arithmetic.

```
Algorithm CountDistinctElems(t1, t2, trace)
  // count distinct elements from t1+1 to t2-1 in trace

  // B: the trace is partitioned in blocks of size B
  // block(l,i): the num. of distinct elems in i'th block of level l,
  //   which contains trace (B^l*i...B^l*(i+1)-1), initially 0
  //   block(0,i) is the same as the trace.
  // integer divisions follows the convention of C language

  // find the reuse distance, reuseDis
  reuseDis = 0
  lvl = 0  // block level
  while (t1/B != t2/B)  // not inside the same block
    reuseDis=reuseDis+block(lvl,t1+1)+...+block(lvl,(t1/B+1)*B-1)
    reuseDis=reuseDis+block(lvl,(t2/B)*B)+...+block(lvl,t2-1)
    t1 = t1/B; t2 = t2/B; lvl=lvl+1
  end while
  reuseDis=reuseDis+block(l+1,(t1/B)+1)+...+block(l+1,(t2/B)-1)

  // maintaining the trace and block counts
  trace(t1) = empty; lvl = 0
  while (t1/B != t2/B)
    t1 = t1/B; block(lvl+1,t1)=block(lvl+1,t1)-1
    t2 = t2/B; block(lvl+1,t2)=block(lvl+1,t2)+1
  end while
End Algorithm
```

Figure 3: Algorithm for counting distinct elements

We now analyze the time and space complexity of the algorithm. Assuming a trace of N data references and a maximal reuse distance of M, the counting for each reference takes at most $O(B)$ if the last two uses happen in the same level-0 block or at most $O(B(logN))$ otherwise. So the overall cost is $O(NBlogN))$. The space cost is $O(N)$.

2.3 Instrumenting Program

We analyze two aspects of a program: the data and their access. We use an instrumentor to insert into programs three types of runtime calls: *RecordData*, *RecordPartialData* and *RecordAccess*. At runtime, the execution of these function calls provides a trace of data structure definition, data allocation and data access. This section describes the construction and insertion of these calls at compile time through an instrumentor. The next section will describe their runtime behavior.

First, the parameters of runtime calls are used to identify the location and structure of data. Since the recording happens at runtime, we have a unique name for each data—its memory address. Then we map from physical memory address to logical data. For a single-element variable, the mapping is one-to-one. For structured data, the mapping is many-to-one. We measure reuse distance for each internal element, but we maintain the link between the element and its parent data structure. For example, we record reuse distance on each array element but also keep track which elements belong to which array. To do so, we record the composition of internal data elements when the data structure is defined. Then at its use, we record the memory address of the containing structure and the address of the accessed element so that we can calculate the offset and locate the data element within the data structure. For example, for array $integer A(5)$, we made the call $RecordData(A, 20, 4)$, which at runtime, gives the base address of array A, the total size in bytes, and the size of each element (assuming 4-byte integers).

After initial data allocation, any access to internal elements of structured data can only be made through the base address of the structured data. For example, any reference to elements of array A must go through the array name, which gives the beginning address of array A. One problem, however, is the extraction of partial data, for example, a partial array. The problem arises when the address of internal elements of structured data is exposed, either explicitly through pointers or implicitly through call-by-reference functions. For example, a C programmer extracts a pointer in the middle of an array, or a Fortran programmer passes a section of the array to a subroutine. Any such action creates a new entry point to the array data. To handle these cases, we use the call, $RecordPartialData$, to record a new entry point. For example, when $A(i,j)$ is passed to a function in Fortran, we insert $RecordPartialData(A, A(i,j))$ to signal that any reference through $A(i,j)$ is a reference to array A.

For each array reference, e.g. $A(e)$, we invoke $RecordAccess(A, A(e))$ to record this access. The inclusion of the base address allows us to calculate the index value e based on the memory address of A, $A(e)$, and previously recorded element size. Note that although textual names are used in these calls, they are physical memory addresses at runtime. When we are interested in statement reordering, we record one statement at a time including all its read and write accesses. In addition, for each statement, we record only access to distinct variables. For example for statement $A(i) = A(i) * 0.2$, we record one access of $A(i)$ instead of two.

The reason we record logical data structures so that data reuse can be described in terms of logical data (e.g. array A), not physical memory (e.g. 0x2001). Another important benefit, as we will see in the next section, is the aggregation of information at runtime. We note that variable aliasing, a difficult problem for compiler analysis, does not exist here because we use runtime memory locations. The textual names of variables do not matter. Also, our scheme can handle union type data if the actual content of the data structure can be discerned based on either static or runtime type indicators. However, we cannot attribute access to different logical data structures if they intersect in physical memory in unknown ways, such as Fortran programs using the storage association feature. In these cases, programmers impose a low-level view of memory, hence reuse distance analysis must treat each memory cell as a single data structure.

Once the runtime calls are constructed, the last issue is where to insert them in a source program. We insert them immediately after type declaration, data declaration and data access but before the next branch statement, so that we do not miss or duplicate any recording. Two cases require special treatment. The first is the single if-statement, e.g. if (*condition*) *statement*. Since the recording adds a new statement, we need to convert single if-statement into blocked if-statement, e.g. if (*condition*) *begin statement*; *recording*; $endif$. The second is the statement of function call. We insert necessary recording statements immediately before the call, including the use of data in case of parameter expressions and the creation of data in the case of call-by-value.

2.4 Runtime Recording

The function calls inserted by instrumentation invoke the reuse distance simulator at runtime. The steps of these calls are described in Figure 4. For each structured data, $RecordData$ records its structure and creates a counter for each of its element. For each new entry into the middle of a structured data, $RecordPartialData$ links this entry to the base entry of the data. Finally, for each memory reference, $RecordAccess$ finds its counter from its parent structure and invokes the counting routine described in Section 2.2.

The main cost of runtime recording comes from the shadow data and hashtable. Shadow data contains both meta data of the structure and a counter for each data element. The amount of meta data is proportional to the number and size of data definitions in the program. In the worst case, it is proportional to the size of the program. This overhead is no more than that stored by a typical debugger. The memory overhead of counters is proportional to the the amount of data elements in the program. In normal programs, shadow data doubles the memory requirement of the program. However, the use of shadow counters is not necessary

```
Algorithm RecordData(base_address, total_size, elem_size)
  create data shadow containing
    1. meta data of the structure
    2. one counter for each internal element
  insert the mapping, base_address-->shadow, in hashtable
End

Algorithm RecordPartialData(base_address, new_entry)
  search hashtable for the shadow of base_address
  insert the mapping, new_entry-->shadow, in hashtable
End

Algorithm RecordAccess(base_address, elem_address)
  search hashtable for the shadow of base_address
  locate the counter in shadow for this element
  call MeasureReuseDistance(elem_address, counter)
End
```

Figure 4: Algorithm for runtime calls

if we do not measure reuse distance with off-line program trace. When we analyze the trace after execution, we can use the space allocated for original program data, except for those whose size is smaller than the size of the counter. In our experiment, the memory overhead is not a serious obstacle because of virtual memory.

The other significant cost comes from the hash table. For each structured data, we need an entry for each of its entry point in hash table. The larger the data structure, the more efficient is the hashing. For example, all elements of an array can be included by a single entry in hash table. In the worst case, we need one entry for each data element. However, we can avoid the overhead of hash table if we do not measure reuse distance with program execution. As mentioned before, the counter for most data elements would be themselves, so there is no problem of finding them, although we still need to attribute element access to its parent structure.

3 Evaluation

3.1 Experimental Setup

We have implemented a reuse distance simulator as described in the previous Section, along with an instrumentor for Fortran 77 programs. The simulator does not yet use a complete tree structure. It merely partitions traces into a sequence of blocks of size 200. The instrumentor is based on the front-end of the D System compiler from Rice University. We are in the process of adding a C and Java instrumentor. The following evaluation, however, measures only Fortran benchmarks, as listed in Table 1.

Name	Source	Lines/Loops/Arrays	Description
ADI	self-written	108/8/3	alternate direction implicit integration
FFT	self-written	92/15/4	fast Fourier transformation
Swim	SPEC95	429/8/15	shallow water model
Tomcatv	SPEC95	221/18/7	mesh generation program
SP	NASA	1141/218/15	computational fluid dynamics (CFD) simulation
Sweep3D	DOE	2105/67/6	wavefront particle transport code

Table 1: Applications tested

3.2 Program Locality

We use the histogram of reuse distances to measure overall program locality. The first graph of Figure 5 shows two curves, each is a histogram of *ADI* with an input size of either 50^2 or 100^2. In each curve, a point at *(x,y)* means that y thousands of memory references have a reuse distance of either 0 when x is 0 or within the range $[2^{(x-1)}, 2^x - 1]$ when $x \geq 1$. We link discrete points into curves to emphasize the elevated hills, where large portions of memory references reside. For example, the first hill of the larger input size at $x = 2$ shows that 80 thousands, or 35% of memory references have a reuse distance of either 2 or 3 elements, while the last hill shows that a similar portion (35%) has a reuse distance between 16K (2^{14}) and 33K ($2^{15} - 1$) elements.

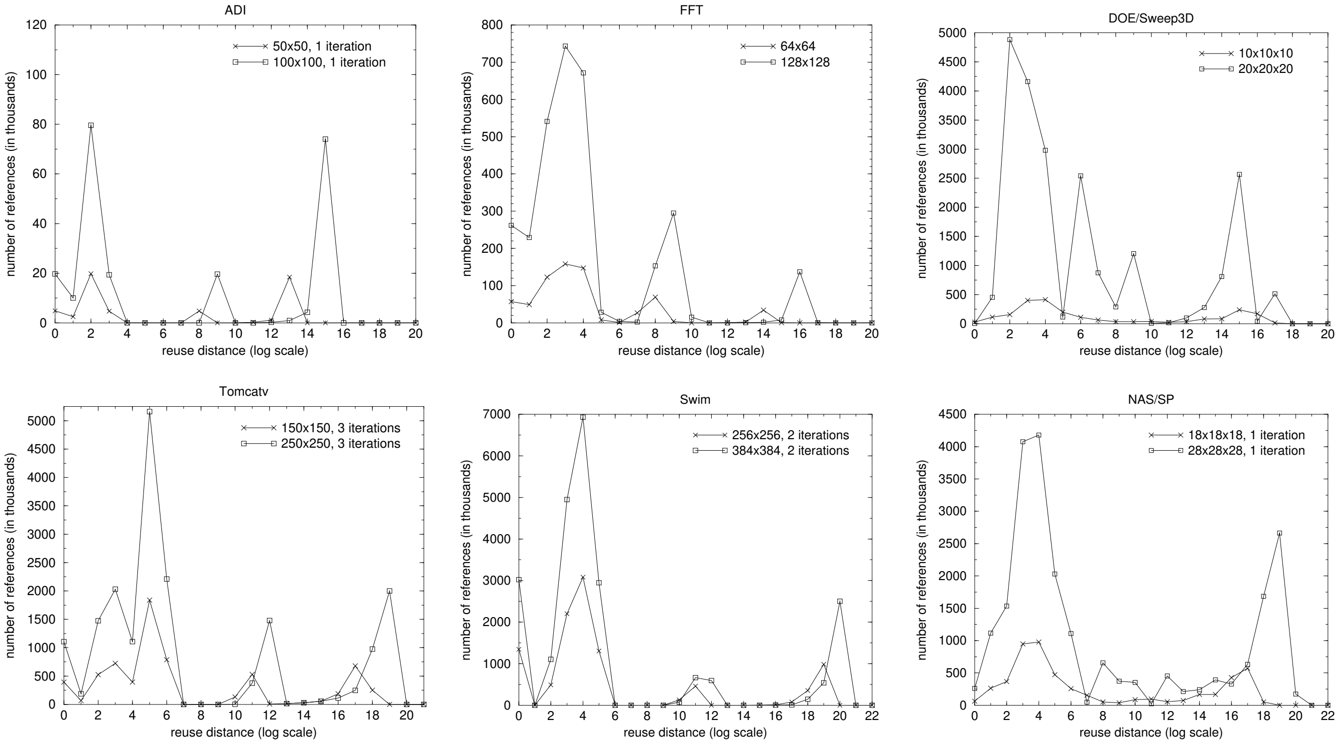


Figure 5: Reuse distances

The remaining parts in Figure 5 show histograms for all other tested programs. Two data sets are used for each program. Like *ADI*, most memory references in these programs fall into a few groups, especially memory references with long reuse distances. Group concentration is high for reuse distances that are longer than 64 elements. For the larger input size of each program, less than 30% of x-axis points contain the majority of long reuse distances: 95% are concentrated at two x-axis points of *ADI*, 96% at three points of *FFT*, and 69% at four points of *SP* and three points of *Sweep3D*.

Because of the high concentration of reuse distances in a few ranges, the performance of a fully associative LRU cache either stays the same or varies greatly from one cache size to another, depending on whether the new size holds the next concentrated range. For example for the larger data size of *ADI*, 20% long distances are between $[256, 511]$ and 74% between $[16K, 33K]$. Assuming that cache is fully associative and with LRU replacement and single-element cache blocks, the two most effective cache sizes are 512 (0.5K) and 32768 (33K). The cache sizes between 0.5K and 33K would make no more than 6% difference in the miss rate compared to the size of 512. Similarly, the effective cache sizes are few for other programs. For another example, *SP*, doubling cache sizes would reduce cache misses by merely 0.2% to 6% unless it reaches the four concentrated ranges, which hold 8% to 32% of long reuse distances.

For set-associative caches, we expect to see a similar relation between miss rate and cache size, although the exact size needed for a particular range may be different. If the results extend to caches on real machines, then analyzing reuse distance is much more useful than simulating a fixed size cache. Since reuse distance analysis estimates the cache performance for all cache sizes, it can help a machine designer to select the best cache size for applications for which the machine is targeted. On machines with reconfigurable cache sizes, the analysis can suggest the best dynamic cache size for each program or for each part of the program.

Another interesting property revealed by Figure 5 is that the shape of histogram curves is remarkablely similar for two input sizes, and that the height difference is almost exactly proportional to the difference in input size. For example for *ADI*, both curves have three hills and the three pairs of hills differ in size in the same proportion as their input sizes. This suggests that the program has three locality groups. The first group, shown by the first hill, has a reuse distance of no more than 8. When the input size increases, the location of the first hill remains the same on the x-axis, although size of the hill increases. The next two groups have longer reuse distances, and the distances increase with the input size. Reuse distances of the second group lengthen in proportion to the square root of input size and the third group lengthens linearly to the input size. We call the memory references in the last two groups *evadable reuses* because they are the cause of cache misses when data size becomes sufficiently large.

The other graphs show similar correlation between the curves of different input sizes. Like *ADI*, the number of hills often corresponds to the different rates of lengthening. This suggests that based on measuring locality groups, we can infer program locality for different input sizes. Although the locality groups are similar for the same application, their composition is very different for different applications. So we cannot estimate an application based on results from other applications. For example, the fastest moving hill contains 35% of all memory references in *ADI* but only 5% in *FFT*. Such variation cannot be accounted by studying a fixed set of benchmarks. Since we need to measure each individual application, reuse distance analysis is a valuable tool because it provides the needed quantitative measurement of program locality.

3.3 Locality Change

In the past, cache optimizations are evaluated by observing the miss rate change on a particular machine. In this section, we measure the effect in a machine-independent manner by analyzing program locality before and after program transformations. At the end, we will also examine the correlation between the change in reuse distances and the change in cache performance on a real machine.

First, we measure the effect of reuse-driven execution [3], which reorders program execution by moving each instruction up the instruction stream and closer to its data-sharing predecessor. In the instrumentation, we record all data accesses of each statement. The change in reuse distances by reuse-driven execution is shown in Figure 6 for a subset of our tested programs. For each program, we measure the histograms of the original program order and the order of reuse driven execution. We tested two input sizes for each program. The change in locality is very similar so we show only the large input size in Figure 6.

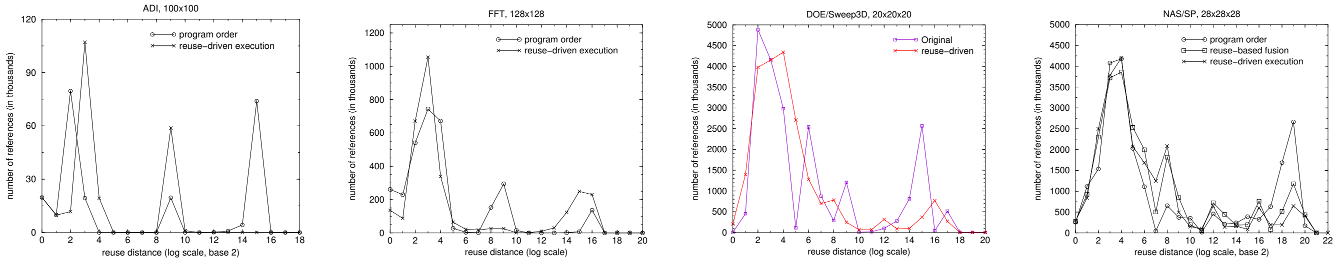


Figure 6: Locality change due to reuse-driven execution

The histograms before and after reuse-driven execution show the overall change in program locality due to the transformation. For *ADI*, the transformation removes the third locality group and reduces the size of the second group. As a result, not only the number of evadable reuses is decreased, the average rate of lengthening is also decreased. These two effects are also seen on *SP* and *Sweep3D*, although in different degrees. Interestingly, the graphs also show that many reuse distances, especially non-evadable reuses, are lengthened as a result of the shortening of evadable reuses. The range of the first hill of *ADI* moves from fewer than 4 elements to fewer than 8 elements after the transformation. *SP* and *Sweep3D* have a new concentration of non-evadable reuses at range $[32, 63]$. Even some evadable reuses are lengthened in *Sweep3D*: a larger hill appears at range $[64K, 128K]$. Furthermore, the transformation is not always beneficial. It has the reverse effect on *FFT*, increasing the length of the evadable reuses.

By comparing program locality, reuse distance analysis shows the complete effect of the transformation that cannot be easily measured by simulating a fixed cache. The change in miss rate varies greatly depending on the choice of cache size. For *Sweep3D* at the larger input size, the reduction in miss rate is 56% for 4K

programs	*ADI*		*FFT*		*Sweep3D*		*tomcatv*		*swim*		*SP*	
input size	50^2	100^2	64^2	128^2	10^3	20^3	150^2	250^2	256^2	384^2	18^3	28^3
time (sec.)	0.38	0.53	0.99	4.29	4.45	59.5	25.7	168	82.2	342	19.8	243
total refs	64.3K	258K	697K	2.85M	2.89M	21.9M	6.76M	19.0M	10.9M	24.5M	5.50M	23.3M
total elems	7.5K	30K	16.5K	65.7K	11.1K	79.6K	155K	433K	464K	1.04M	212K	811K

Table 2: Time needed for reuse distance analysis

cache, becomes 11% higher (62% reduction) for 8K cache, then decreases for 16K and 32K cache (61% and 54%), and finally changes to an 86% increase in miss rate for 64K cache. Such global picture is not possible to attain by simulating single-size cache. Furthermore, cache simulation would not reveal the unusual increase in short reuse distances. For both *SP* and *Sweep3D*, a large hill appears at range $[64, 127]$. This result has important implication to the top level of memory hierarchy—registers.

The *SP* figure at the larger input size contains a third curve, which is the histogram of a source-level transformation—*reuse-based loop fusion* [3]. Its comparison with the first curve shows the locality improvement over the original program, while its comparison with the second curve shows the unrealized potential. Source-level fusion reduces the number of evadable reuses by 45% while runtime trace reordering reduces it by 63%. Since source-level fusion can be implemented and measured on a real machine, we tested it for class B input on a processor on SGI Origin2000. We observed a reduction of 51% in L2 misses, remarkably closer to the 45% reduction predicted by reuse distance analysis. The fusion curve has a large hill at range $[64, 127]$, which signals trouble for registers because most machines have fewer than 64 registers. Indeed, for class B input, loop fusion increased the number of register loads and stores by 20% on SGI Origin2000. The results from *SP* strongly suggest that reuse distance analysis is effective in predicting memory hierarchy performance on real machines for large applications, although additional study is needed to verify the accuracy of the prediction.

3.4 Measurement Time

We measured the analysis time on a 250MHz R12K processor of SGI Origin2000. Table 2 lists, for each application at each input size, the simulation time, the total number of memory references and the total number of distinct data elements. The first row is the wall-clock time measured by the Unix *time* utility. The second row is the total number of memory references analyzed by the simulator. The third row is the total number of distinct data elements that have at least one access by the program. As mentioned before, the current simulator does not use a tree algorithm. Instead, it partitions the trace into a sequence of blocks of size 200.

The *time* row of Table 2 shows that three analyses take less than one second; five take more than a second but less than a minute; and the remaining ones take up to six minutes. The next two rows show that higher simulation time is a result of more memory references and larger data size. The simulation analyzes up to 700 thousand references and 30 thousand data elements in one second, up to 22 million references and 200 thousand elements in one minute, and up to 25 million references and one million elements in six minutes. The execution time correlates with the data size more than it does with memory references. Two analyses have a similar ratio of memory references to data size: the larger size of *SP* and the smaller size of *Swim* have on average about 30 references per data element. The former takes 20 seconds on 212 thousand elements, and the latter takes 17 times as long on five times as much data, indicating cache performance plays a critical role in simulation time, just as it does in program execution time. The larger size of *Swim* has more than a million double numbers (8MB), much larger than the level-two cache on the processor (4MB). Although not listed in the table, we measured the amount of resident memory reported by the Unix *top* facility. For the larger input size, resident memory is no more than 44MB for *Tomcatv*, 55MB for *Swim*, 78MB for *SP* and much smaller for other runs. The result table does not show the simulation for reordered programs used in the previous section. In general, the simulation takes less time because of the shortening of reuse distance by the optimization. For the larger size of *SP*, the version after loop fusion takes 56simulate than the original version.

In summary, the time and space cost of reuse distance analysis is reasonable for the input sizes we used. Although up to 25 million accesses to over one million data elements were simulated, all analyses completed within 6 minutes and required no more than 80MB physical memory.

4 Related Work

Modeling cache performance has a long history. The most often used technique is trace-driven simulation, which collects the trace of program memory accesses and runs it through a cache simulator. In 1970, Mattson et al. published a stack algorithm, which, for caches using a set of replacement policies including LRU, can measure cache miss rate for cache of all sizes in one pass [5]. For LRU replacement, the stack algorithm keeps all accessed data in a stack. For each data element, the stack position or level is the number of distinct elements that have been accessed after the last access of the data. In other words, it is the reuse distance between each element and the current access. The stack algorithm consists of two steps: stack search to find the stack level for the current access, and stack update to change the stack levels of other elements after the new access. Because of the need to search and update potentially a large number of stack elements, stack algorithms are expensive when the number of data elements and the length of their reuse distances are large.

In 1975, Bennett and Kruskal observed that the miss rate of fully associative LRU cache depends on the same concept as what we call reuse distance. They developed the counting algorithm that we have described in Section 2.2. When simulating the paging performance for database applications, they observed a reduction in execution time from several hundred seconds to a few seconds. One significant overhead, however, is the mapping from each data element to the time of its last access. Bennett and Kruskal used a hash table that is at least as large as the size of all data. They also proposed to separate this cost from the counting by going through the trace multiple times. In this work, we used the same algorithm for simulation but reduced the demand for hash table by including the information of source-level data structure through compiler instrumentation. For example, instead of creating a hash entry for each array element, we insert only one entry for the whole array and compute the offset without further lookup. An important limitation of the counting algorithm, compared to the stack algorithm, is that it is not yet flexible enough to model multi-configurations such as different cache line sizes, replacement policies, set associativity, and writebacks.

Several later studies have improved the efficiency of the stack algorithm. For the search step, Olken used an AVL tree to organize stack elements and the element level can be computed with the tree search [7]. Observing that lack of locality in AVL tree operations, Sugumar and Abraham [10] used self-adjusting binary search tree [8]. The cost of stack update has been harder to reduce. Several methods exploited the reference locality, i.e. the new reference is likely to access an element that is recently accessed. Smith removed such references from the trace [9]. Sugumar and Abraham monitored them separately from other stack elements [10]. They observed that their method worked well for programs with good locality but was still expensive for others. Several studies compared the variations of the stack algorithm with the counting algorithm [1, 7, 11, 10]. Since different machines and different implementations were used, the quantitative results were not directly comparable. All former studies simulated traces that are much smaller than those we used here. In the most recent one by Sugumar and Abraham, the traces contained at most 10K elements and 500K references and took more than six minutes on Sun 3/60 in the longest run. Many studies have extended the stack algorithm to model different cache configurations at the same time. This topic is beyond the scope of this report.

Reuse distance analysis goes one step beyond trace-driven simulation because it keeps the structure of program data. For each reference to a data element, it keeps track of the address of the parent structure and the organization of the structured data. Therefore, the program locality can be measured for data structures and their elements. The structure information is also vitally important in reducing the size of hash table needed when performing analysis with the execution.

Instrumentation with source-level information is not a new idea. Most of today's compilers include source-level annotations in their generated code. Typical debuggers also include the type (structure) information with data. Till the recent past, most cache analysis tool includes only the source information about the program flow, i.e. loop nests and functions, not its data structure. To analyze data, Ding instrumented array access with its array structure information [2]. Building on this idea, Mellor-Crummey et al. developed a web-based tool, $MHSIM$, that attributes cache misses to both source-level control and data structures [6]. We plan to integrate our instrumentation and analysis into $MHSIM$.

5 Conclusion

In this report, we model program locality by measuring the distance of all its data reuses. We have described a simulator and an instrumentor. The former provides fast analysis for full-size applications with large data sets, using the counting method given by Bennett and Kruskal in 1975 [1], and the latter ascribes the simulation result to source-level data structures, using a novel compiler instrumentation. With this tool, we have analyzed data reuse behavior in a set of benchmark applications. The main findings are:

- Most memory references fall into a few locality groups whose reuse distances stay constant or change predictably with input size.
- Cache enhancing transformations reduce the length of long reuse distances but often lengthen short reuse distances.
- The change in evadable reuses corresponds well with the change in the actual change of miss rate in a large application.
- Reuse distance analysis is fast in practice, analyzing up to 25 million memory references on one million data elements in less than six minutes.

Acknowledgment

We thank D System group at Rice University for providing the SGI Origin machine used in this experiment.

References

[1] B.T. Bennett and V. J. Kruskal. LRU stack processing. *IMB Journal of Research and Development*, pages 353–357, 1975.

[2] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Dept. of Computer Science, Rice University, January 2000.

[3] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proceedings of International Parallel and Distributed Processing Symposium*, San Francisco, CA, 2001. http://www.ipdps.org.

[4] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[5] R. L. Mattson, J. Gecsei, D. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[6] J. Mellor-Crummey, R. Fowler, and D. B. Whalley. On providing useful information for analyzing and tuning applications. In *Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems*, Cambridge, MA, 2001.

[7] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.

[8] D. D. Sleator and R. E. Tarjan. Self adjusting binary search trees. *Journal of the ACM*, 32(3), 1985.

[9] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, 3(1), 1977.

[10] R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan, 1993.

[11] J. G. Thompson. *Efficient analysis of caching systems*. PhD thesis, University of California, Berkeley, 1997.