Reuse Distance Analysis for Scientific Programs

Yutao Zhong[1], Chen Ding[1], and Ken Kennedy[2]

[1] Computer Science Department, University of Rochester, Rochester, New York, USA
[2] Center for High Performance Software, Rice University, Houston, Texas, USA

Abstract. Cache is widely used in today's computing systems and plays a pivotal role in system cost, performance and energy consumption. Most programs benefit from cache because they have locality. Until now, program locality has been measured by precise but expensive cache simulation, which has to be repeated for every cache size and program input. In this paper, we measure program locality by the distance between data reuses. Reuse distance is an inherent program property. With a single measurement, one can quantitatively analyze the locality of individual program data as well as an entire application. We make three contributions in this work. First, we give an approximation method for measuring reuse distance. The new method reduces the space overhead by orders of magnitude compared to existing techniques. Second, we use compiler instrumentation so that the results of the analysis apply to the program source, rather than the compiled binary. Finally, we measure an extensive set of scientific applications and discuss their program locality.

1 Introduction

The performance of modern machines directly depends on the performance of cache. In addition, cache plays a critical role in system energy consumption because it occupies a large portion of CPU chip space (up to 92% [2]) and determines the way that memory is utilized. The effectiveness of cache hinges on a term called program locality, which measures how well a program can cache its data accesses. For regular programs, compiler analysis can determine the exact sequence of data accesses. For other programs, cache simulation can determine the miss rate by simulating them on a particular cache. Since cache simulation does not predict performance for unsimulated cache sizes and program inputs, it does not tell us the full impact of program locality.

In this paper, we measure program locality by a concept we call *reuse distance*. Given a sequence of data accesses, reuse distance is the number of distinct data elements accessed between two consecutive uses of the same element. Figure 1 shows an example sequence and the distance between each pair of data reuses. For example, the distance between the reuses of b is two because two distinct data elements, a and c, are accessed in between. Reuse distance is identical to stack distance defined by Mattson et al [10], who measured the reuse of virtual memory pages with a stack algorithm. We call it reuse distance here because it models the reuse of program data and can be measured by much faster methods that do not use a stack. Reuse distance does not measure exact cache performance; however, it does determine the number of capacity misses for all cache sizes [7].

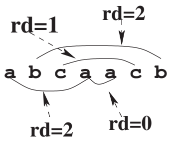


Fig. 1. Reuse Distance

Being purely a property of programs, reuse distance has at least two important uses. First, we can measure and compare reuse distance of different programs or different data items within the same program. Reuse distance, which is per each data reference, provides much richer information than the miss rate, which is per program and is meaningful only for a fixed cache size. With reuse distance, we can evaluate cache optimization in more detail by measuring changes in different data structures. We can separate program-specific factors from machine-specific factors and therefore manage inherent program behavior separately from its interaction with a particular machine.

Reuse distance also helps a program to find the best cache size by giving the number of its capacity misses for all cache sizes. Caches have different sizes, associativities, and hierarchies in different machines. In a heterogeneous system such as computation grid [6], one improves system utilization by fitting each program with a machine that has a suitable memory hierarchy. Cache selection becomes even more useful on recently proposed adaptive cache, which provides a program with a much wider range of choices [1].

In this paper, we study and answer three questions. The first is how to measure reuse distance with reduced time and space costs. We will present an approximation algorithm that greatly reduces space cost by allowing a constant bounded error in the result. The second question is how to present reuse distance results to a programmer. We will use a compiler-assisted system to measure and report reuse distance for each source-level program data item. The last question is what is the locality of commonly used programs. We will measure an extensive set of scientific benchmarks. We will also show applications in program analysis and optimization. The rest of this paper is organized as follows. Section 2 describes the approximation algorithm and the compiler support. Section 3 reports the results obtained for a set of scientific programs. Section 4 reviews related work and, finally, Section 5 concludes.

2 Reuse Distance Measurement

We first describe an approximation method for measuring reuse distance and then a compiler-assisted tool for supporting the measurement in programs.

2.1 Approximation Method

Accurate analysis of reuse distance carries a significant space overhead. It either stores the entire access trace [3] or stores all program data in a stack [10] or in a tree [12, 15]. In fact, the space requirement of all current accurate analysis is at least M, which is the size of program data. The high space overhead makes accurate analysis unfit for use in run-time or on-line optimization because the analysis part itself uses as much memory as the analyzed program. In this section, we describe an approximation method that incurs a tiny fraction of the memory at the cost of accurate analysis. In other words, we trade accuracy for efficiency.

The approximation method follows the general approach of counting but it does not store the whole trace of memory references. Given a block size B, it divides the trace into two parts: the *precise trace*, which includes the latest B distinct elements that have been accessed; and the *approximate trace*, which includes an approximate summary of all earlier references. The measurement is accurate for reuses in the precise trace and the error for reuses in the approximate trace is bounded by the constant B. The algorithm is shown in Figure 2.

Both precise and approximate traces are organized as a B+-tree. We call the tree for the precise trace *P-tree* and the approximate trace *A-tree*. In P-tree, each leaf records the last access time of a distinct data element. Each tree node records the total number of references contained by its children. In A-tree, each leaf represents a time range and the number of data elements whose latest accesses are inside that range. The main method, *CountDistinctElems*, takes in the previous access time, t_1, determines which tree t_1 is in, and searches the tree node containing t_1. The reuse distance is calculated once the tree node is found. The algorithm then updates the traces. The new reference at t_2 becomes the last access, replacing t_1. The algorithm either deletes the node of t_1, if it is in P-Tree, or decreases the element count in the range containing t_1 in A-Tree. In both cases, the new reference time t_2 is added to the beginning of P-Tree. If P-Tree exceeds the size of $2 * B$, it will split at the root and one half of it will be passed to A-Tree as a new leaf node.

The space saving happens in the approximate trace. Each leaf of A-Tree represents a trace segment that contains between $\frac{B}{2}$ to B elements whose latest access time falls inside the segment. Standard B-Tree maintenance methods are used to merge and split tree nodes whose element count falls below $\frac{B}{2}$ or rises above B. The upper bound ensures that the error of reuse distance measurement is no more than B. The lower bound ensures space saving. Since at least half of the leafs of A-Tree contain no fewer than $\frac{B}{2}$ distinct elements, the total number of leafs is no more than $\frac{4M}{B}$ and the tree size is no more than $\frac{cM}{B}$, where M is the number of elements in a program and c is a small constant. The space reduction is roughly a factor of B, which is usually in the order of thousands.

The transfer from precise to approximate trace happens when the number of elements in P-Tree exceeds $2B$, as shown in the last part of Figure 2. The function *TreeSplit* bisects P-Tree. It should preserve two properties: the size of remaining part is no less than B, and the size of the split part is no less than $\frac{B}{2}$. The first ensures that P-tree is valid after splitting, i.e. it measures accurately

```
Algorithm CountDistinctElems(t1, t2, trace)
  // this method counts distinct elements from t1+1 to t2-1 in trace
  trace = (P-tree, A-tree)  // precise and approximate trace
  tp: the time of the earliest accessed element in P-tree

  // calculate reuse distance
  if (t1 > tp) // search precise trace for last access
    reuseDis = TreeSearchAndCount(t1, P-Tree)-1
    TreeDelete(t1, P-Tree)
  else         // search approximate trace for last access
    reuseDis = TreeSearchAndCount(t1, A-Tree)
    TreeDelete(t1, A-Tree)
  end if

  // insert latest reference
  TreeInsert(t2, P-Tree)

  // split P-Tree if it becomes too large
  if (size of P-Tree is greater than 2*B)
    newApproxBlock = TreeSplit(P-Tree)  // tp is adjusted accordingly
    TreeInsert(newApproxBlock, A-tree)
  end if
End CountDistinctElems

Algorithm TreeSplit(P-Tree)
  // Cut off at least B/2 nodes while maintaining a valid P-Tree
  totalChildren = 0
  for each cell c in the root node of P-Tree, from the latest
    totalChildren += children of c
    if (totalChildren > B)
       remove all earlier cells of c
       collapse the deleted cells into a new block N
       return N
    end if
  end for
End TreeSplit
```

Fig. 2. Approximation algorithm

for reuses whose distance is no more than B. The second ensures that the split part is a valid leaf for A-tree and the frequency of splitting is lower than one for every $\frac{B}{2}$ memory references. In fact, a tree split will not happen unless the program has accessed at least $\frac{B}{2}$ data elements after the last split.

An efficient implementation of function $TreeSplit$, as shown in Figure 2, bisects P-Tree at the root level. For it to observe the two properties described in last paragraph, we need to carefully choose the block size of P-Tree with respect to B, the block size of A-tree. Let the block size of P-Tree be B_p. Suppose the maximal height of P-Tree is k, counting from 0. Each child of the root node contains minimally $(B_p/2)^{(k-1)}$ and maximally $B_p^{(k-1)}$ elements. We need to have $(B_p/2)^{(k-1)}(B_p) \geq 2B$ and $B_p^{(k-1)} \leq (B/2)$. The first inequality ensures that the tree height is at most k at the time of splitting, and the second guarantees the split happens at the root level. For example, when B is 2000 and k is 3, the function $TreeSplit$ will work as desired when B_p is 32.

2.2 Program Instrumentation

We analyze the accesses to data at the source-level. To do so, we insert into programs three types of runtime calls: *RecordData*, *RecordPartialData* and *RecordAccess*. At runtime, the execution of these function calls extracts structure definition and the trace of data allocation and data access. This section describes an instrumentor that constructs and inserts these calls at compile time. The next section will describe their runtime link to the measurement algorithm.

Since the recording happens at runtime, we have a unique name for each data item—its virtual memory address. But we need to map from virtual memory address to source-level data. A hash table is used for this purpose. For a single-element variable, the mapping is one-to-one. For structure or array data, the mapping is based on the starting address of the structure or array. So we convert many-to-one mapping into one-to-one. For example for an array, we record its starting address and element size. Assuming an integer array $A(5)$, we insert a call to $RecordData(A, 20, 4)$, which at run-time, records the base address of array A, the total size in bytes, and the size of each element.

For each data access to a structure or array, we need to find its starting address and then find the element by its offset in the array. The problem is easy if data elements are accessed through their base, i.e. accessing A elements through the array name. In such cases, this base address is used as the first argument of function call *RecordAccess*. For instance, for array reference $A(e)$, we insert a call $RecordAccess(A, A(e))$. The second argument is the memory address of the element. Their difference is the byte offset of the accessed element in the structure or array. Because we have recorded its composition, we can calculate the element offset from the byte offset. For an array, the element offset is the byte offset divided by element size. Note that although textual names are used in these calls, they are virtual memory addresses at runtime.

One complication, 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. Then, we can use exactly the same format of $RecordAccess$ to record references to partial data. The only requirement is to let the first argument passed to $RecordAccess$ denote an entry point of an array.

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 accesses to distinct variables. For example for statement $A(i) = A(i) * 0.2$, we record one access to $A(i)$ instead of two.

The instrumentation allows us to describe data reuse of source-level data, e.g. array element $A(20)$ instead of memory address $0x2001$. Another important benefit, as we will see in the next section, is the aggregation of information. 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 although it cannot always distinguish the source-level fields if they have the same byte offset. We assume input programs are type safe, that is, different data do not intersect in physical memory. Fortran programs that use the storage association feature violate type safety. In such case, programmers impose a low-level view of memory, hence reuse distance analysis must treat the entire memory as one byte array.

Our instrumentor inserts run-time calls 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 a 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 a function call. We insert necessary recording statements immediately before the call, including the use of data accessed in actual parameters. For efficiency and simplicity, we do not record local data and their accesses. We do not expect a large amount of local data, so their impact on reuse distance is inconsequential, especially for long reuses.

2.3 Runtime Recording

The function calls inserted by the instrumentor invoke the reuse distance analyzer at runtime. The steps of these calls are described in Figure 3. For each variable, $RecordData$ records its structure and creates its *shadow data*, which consists a counter for each of its elements. When the address is taken from the middle of a structure data, $RecordPartialData$ links this entry to the starting address of the data. Finally, for each memory reference, $RecordAccess$ finds its counter from its containing structure and invokes the counting routine described in Section 2.1.

```
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
```

Fig. 3. Algorithm for runtime calls

The main cost of runtime recording comes from the shadow data and hash table. Shadow data contains both meta data of the structure and a counter for each data element. The amount of meta data is proportional to data definitions in a program. In the worst case, it is proportional to the size of the program. The overhead is no more than that stored by a typical debugger. The memory overhead of the counters is proportional to the amount of data elements in the program. In normal programs, shadow data double the memory requirement of the program. The space cost is smaller if we analyze the off-line program trace and do not store the actual program data. In our experiment, the memory overhead is not a serious problem because of virtual memory.

The other significant cost comes from the hash table. Since we index each structure data by its starting address, the hash table needs only one entry for, for example, an array of one million elements. However, taking address from the middle of an array will add more entries to the hash table. In the worst case, we need one entry for each data element.

3 Evaluation

3.1 Experimental Setup

We have implemented both Bennett and Kruskal's [3] and the approximation algorithm as described in section 2.1, along with an instrumentor for Fortran 77 programs. The approximate counting analyzer uses B+-tree with block size 2000. The instrumentor is based on the D System Fortran compiler from Rice University. We have implemented the recording of scalars and arrays as well as the handling of branches and function parameters. Our implementation correctly instruments all tested applications, shown in Table 1. The results in the remaining parts were collected on a 250MHz R12K processor of an SGI Origin2000.

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
Sweep3D	DOE	2105/67/6	wavefront particle transport code
SP	NASA	1141/218/15	computational fluid dynamics (CFD) simulation
Swim	SPEC95	429/8/15	shallow water model
Tomcatv	SPEC95	221/18/7	mesh generation program
Hydro2d	SPEC95	4227/163/6	galactical jets model
Mgrid	SPEC95	666/57/5	3D potential field

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 4 shows two curves of *ADI* for input size 50^2 and 100^2. In each curve, a point at *(x,y)* denotes the percentage (y) of the total memory references that have a reuse distance of either 0 when x is 0 or within the range $[2^{(x-1)}, 2^x - 1]$ when $x \geq 1$. The graph links discrete points to emphasize the peaks, where large portions of memory references reside. For example, the first peak of the smaller input means that nearly 35% of memory accesses have a reuse distance of at most 2^3, or 8. Therefore, with an 8-element cache, these 35% of total memory references, which would be approximately 20,000 memory references, will not incur capacity misses. However, the remaining 65% will.

The remaining part in Figure 4 shows 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 peaks, 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% of long reuses are 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 effectiveness of cache is highly sensitive to only a few sizes. For example in *ADI* with the larger input, 95% of references with reuse distances greater than 8 are concentrated in two peaks: $[256, 511]$ and $[16K, 33K]$. 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

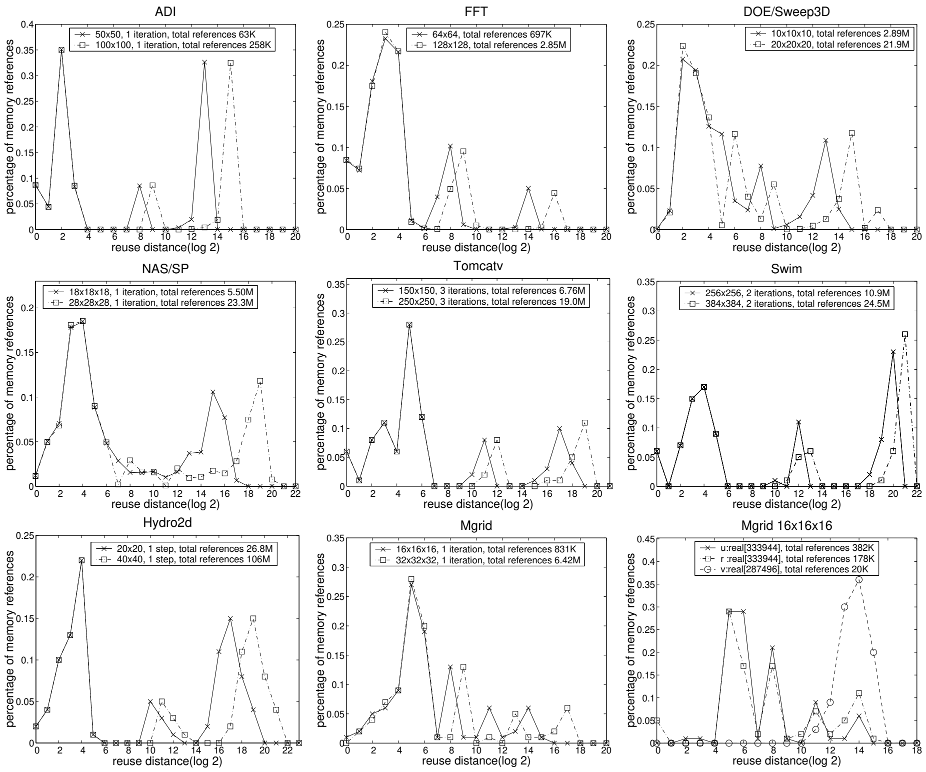


Fig. 4. Reuse distances

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 size would reduce cache misses by merely 0.2% to 6% unless it reaches the four concentrated ranges, which hold 8% to 32% of references whose reuse distance is greater than 64.

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 is that histograms of different input sizes have similar shapes. The two histograms of *ADI* show interesting correlation between reuse distances and data input sizes. In the graph, the larger input is four times as large as the smaller input. Both histograms have three peaks. All three peaks have the same value on y-axis, which means the concentration distribution of the program is an inherent property independent of the input. The position of the first peak is the same for both inputs but the other two peaks change with increases in the size of the input. Reuse distances are doubled in the second peak and quadrupled in the last peak. The relation actually holds for all input sizes, as our compiler analysis confirms. So we can select the best cache size for any input size with a single histogram.

In *ADI*, reuses in the last two peaks have longer distances, and the distance increases with the input size. We call the memory references in the last two peaks like this *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 peaks often correspond 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 peak 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.

The last graph illustrates that different source-level data of the same program may have different reuse behaviors. Therefore different variables in a program can be processed differently according to their distinguished reuse pattern.

3.3 Locality Change

In the past, cache optimizations are evaluated by observing the change of miss rate on a particular machine. In this section, we measure the effect by analyzing program locality before and after program transformation, without any presumption about hardware. At the end, we will also examine the correlation between the change in reuse distances and the change in miss rate on a real machine.

First, we measure the effect of reuse-driven execution [5], which reorders program execution by moving each instruction up the instruction stream and closer to its data-sharing predecessor. The change in reuse distances by reuse-driven execution is shown in Figure 5 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 only show the larger input size in Figure 5.

The histograms before and after reuse-driven execution show the overall change in program locality due to the transformation. There are two effects: the number of evadable reuses is decreased and the average rate of distance increase is also decreased. They can be seen on *SP* and *Sweep3D*, although in

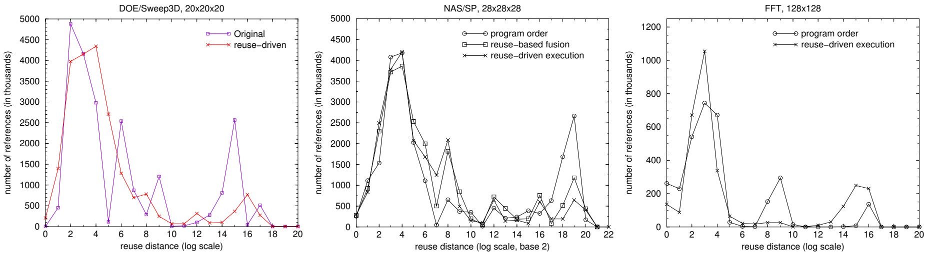


Fig. 5. Locality change due to reuse-driven execution

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. *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 peak 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 cache, becomes 11% higher (62%) 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. Furthermore, cache simulation may not reveal the unusual increase in short reuse distances. For both *SP* and *Sweep3D*, a large peak appears at range $[64, 127]$. This result has important implication to registers.

The *SP* figure at the larger input size contains a third curve, which is the histogram of *reuse-based loop fusion* [5]. 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%. We tested it for class B input on a processor on SGI Origin2000 and 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 peak 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 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 prediction accuracy.

3.4 Measurement Time

Table 2 lists, for a subset of tested applications, the simulation time, the total number of memory references and the total number of distinct data elements. To save space, we omit results for *ADI* and *Tomcatv*, which required an execution time of 0.53 and 168 seconds respectively using the larger input. The execution time is collected by precise counting method due to Bennett & Kruskal [3] with block size 200. The precision loss and efficiency gain of the approximation algorithm is shown in Figure 6. With a loss of accuracy bounded by 2000, we achieve a speedup of 1.55∼1.99.

programs	*FFT*		*Sweep3D*		*SP*		*Swim*		*Mgrid*		*Hydro2D*	
input size	64^2	128^2	10^3	20^3	18^3	28^3	256^2	384^2	16^3	32^3	20	40
time (sec.)	0.99	4.29	4.45	59.5	19.8	243	82.2	342	2.76	17.5	90.0	370
total refs	697K	2.85M	2.89M	21.9M	5.50M	23.3M	10.9M	24.5M	831K	6.42M	26.8M	106M
total elems	16.5K	65.7K	11.1K	79.6K	212K	811K	464K	1.04M	161K	1.06M	4.56M	18.0M

Table 2. Analysis time of reuse distance analyzer

The *time* row of Table 2 shows simulation times ranging from less than one second to a little over six minutes. The slowdown of the execution for Hydro2D with input 40*40, which has the largest number of total references is 235∼250 times compared with the original program. 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 56% less time to simulate than the original version.

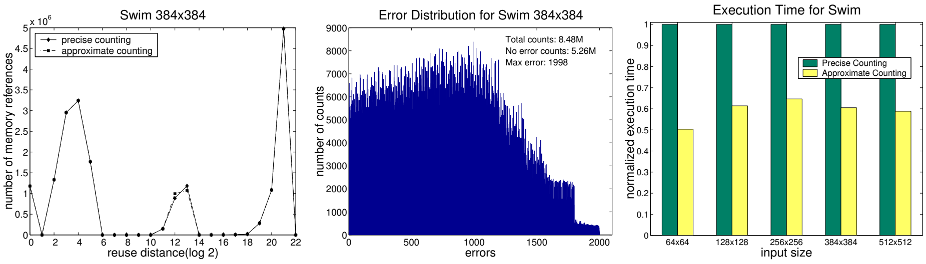


Fig. 6. Precision and efficiency of accurate and approximate counting.

In summary, the time and space cost of reuse distance analysis is reasonable for the input sizes we used. Up to 106 million accesses to over 18 million data elements were simulated. Even with precise counting method from Bennett & Kruskal, all analyses completed within 6.2 minutes and required no more than

140MB physical memory. Our experiments show that the approximation method can decrease the memory usage and also the execution time accordingly.

4 Related Work

Measuring program locality has been an active area in compiler research. Many effective techniques have been developed. However, not all programs are amenable to compiler analysis. Examples include programs with recursion and data references through indirections. Our goal is to provide a general method to measure locality in all programs.

Modeling cache performance has a long history. In 1970, Mattson et al. published a stack algorithm, which, for caches using a set of replacement policies including LRU, can measure the cache miss rate for cache of all sizes in one pass [10]. The idea is to keep data in a stack from most to least recently accessed. The stack algorithm is not efficient for long reuses. For each access, the analysis may need to scan through all stack data and may need to update their locations as well. To find a faster alternative, Bennett and Kruskal developed the counting algorithm that we also implemented to compare with our approximation method. Bennett and Kruskal used a hash table that was as large as the size of all data [3]. In this work, we used the same algorithm for simulation but reduced the demand for hash table by aggregating source-level data through compiler instrumentation. The counting algorithm cannot model multi-configurations such as different cache line sizes, replacement policies and set associativity, because it does not scan through the trace at each memory access.

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 [12]. Observing that lack of locality in AVL tree operations, Sugumar and Abraham [15] used a self-adjusting binary search tree [13]. The cost of stack update is 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 [14]. Sugumar and Abraham monitored them separately from other stack elements [15]. They observed that their method worked well for programs with good locality but was still expensive for long reuses. Several studies compared the efficiency between the stack algorithm and the counting algorithm and found different results [3, 12, 16, 15].

The efficiency of the stack algorithm can be improved by relaxing the accuracy requirement. Kim et al. described a method that measures the logarithm of stack distance to the base of two in $O(N * logM)$, where N is the number of memory references and M is the number of distinct data elements [8]. In comparison, our approximation algorithm has similar time efficiency but lower space cost and higher accuracy.

All aforementioned studies used trace-driven simulation, which collected the trace of binary memory accesses and ran it through a cache simulator. Our analysis is different because it uses a compile-time instrumentor and measures on source-level data, not program binary. The structure information of source-level data is vitally important in reducing the size of hash table needed by the run time analyzer. We analyzed much larger volumes of data accesses than previous studies. 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, while ours took about the same time on a faster machine for traces that were 100 times as large.

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 tools included 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 [4]. Li et. al. defined reuse distance on values but not on locations [9]. They used short reuse distances to study register allocation, but they did not measure long reuse distances. Mellor-Crummey et al. developed a web-based tool, *MHSIM*, that attributes cache misses to both source-level control and data structures [11]. We plan to integrate our instrumentation and analysis into *MHSIM*.

5 Conclusion and Future Work

In this paper, we modeled program locality by measuring the distance of its data reuses. We have described an analyzer and an instrumentor that can analyze general Fortran programs. The former provides fast analysis for full-size applications with large data sets, and the latter ascribes program locality in terms of source-level data. With this tool, we have studied data reuse behavior in representative scientific programs. The main findings are:

- Memory references of most programs fall into a few locality groups whose reuse distances stay constant or change predictably with input size. For these programs, only a few cache sizes matter.
- Reuse distance analysis can find critical cache sizes for different input sizes. So it promises fast and accurate cache-size selection, which is important in resource utilization in heterogeneous and adaptive computing environments. In addition, it can measure locality changes due to program transformation.
- Reuse distance analysis is fast in practice, analyzing up to 25 million memory references on one million data elements in less than six minutes.

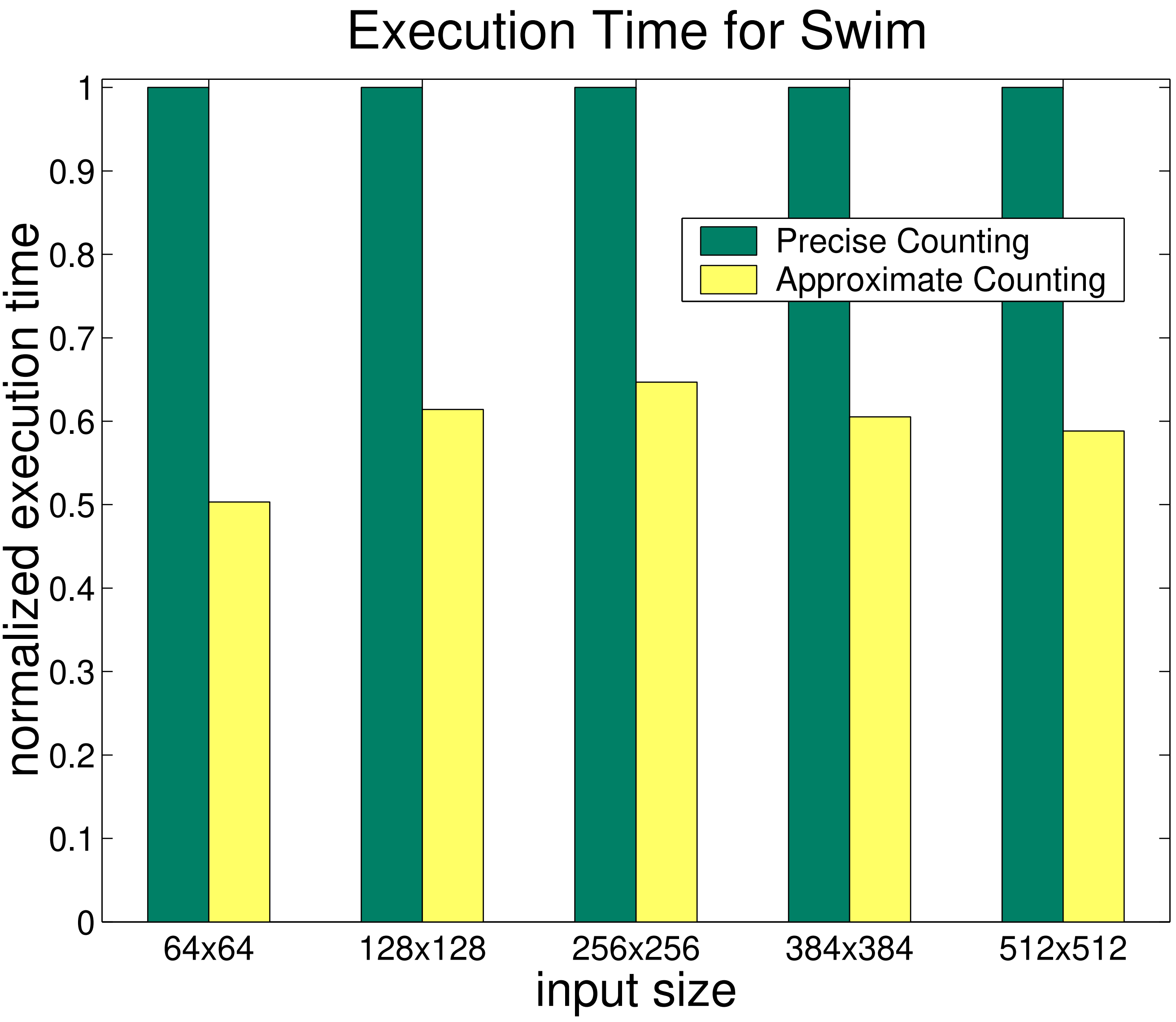
We are expanding the current framework to analyze C programs. At the same time, we are evaluating the accuracy and efficiency of our approximation algorithm more thoroughly. We are also developing a new model of spatial locality. The framework is being used to study compiler or run-time locality-enhancing optimization.

Acknowledgment

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

References

1. R. Balasubramonian, D. Albonesi, A. Buyuktos, and S. Dwarkadas. Dynamic memory hierarchy performance and energy optimization. In *Proceedings of 33rd International Symposium on Microarchitecture*, December 2000.
2. P. Bannon. Alpha 21364: A scalable single-chip SMP. *Microprocessor Forum*, October 1998.
3. B.T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, pages 353–357, July 1975.
4. C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse.* PhD thesis, Dept. of Computer Science, Rice University, January 2000.
5. 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, April 2001. http://www.ipdps.org.
6. I. Foster and C. Kesselman (eds.). *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, 1999.
7. M. D. Hill. *Aspects of cache memory and instruction buffer performance.* PhD thesis, University of California, Berkeley, November 1987.
8. Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 212–213, May 1991.
9. Z. Li, J. Gu, and G. Lee. An evaluation of the potential benefits of register allocation for array references. In *Workshop on Interaction between Compilers and Computer Architectures in conjuction with the HPCA-2*, San Jose, California, February 1996.
10. R. L. Mattson, J. Gecsei, D. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
11. J. Mellor-Crummey, R. Fowler, and D. B. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.
12. F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.
13. D. D. Sleator and R. E. Tarjan. Self adjusting binary search trees. *Journal of the ACM*, 32(3), 1985.
14. A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, 3(1), Janurary 1977.
15. R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan, August 1993.
16. J. G. Thompson. *Efficient analysis of caching systems.* PhD thesis, University of California, Berkeley, 1997.


Execution Time for Swim
normalized execution time
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Precise Counting
Approximate Counting
64x64
128x128
256x256
384x384
512x512
input size