Memory Distance Measurement for Concurrent Programs

Hao Li, Jialiang Chang, Zijiang Yang and Steve Carr

{hao.81.li|jialiang.chang|zijiang.yang|steve.carr}@wmich.edu
Western Michigan University, Kalamazoo MI, U.S.A.

Abstract. Memory distance analysis, the number of unique memory references made between two accesses to the same memory location, is an effective method to measure data locality and predict memory behavior. Many existing methods on memory distance measurement and analysis consider sequential programs only. With the trend towards concurrent programming, it is necessary to study the impact of memory distance on the performance of concurrent programs. Unfortunately, accurate measurement of concurrent program memory distance is non-trivial. In fact, due to non-determinism, the reuse distance of memory references may differ with the same input set across multiple runs. Since memory distance measurement is fundamental to analysis, we propose a measuring approach that is based on randomized executions. Our approach provides a probabilistic guarantee of observing all possible interleavings without repeated executions. In order to evaluate our approach, we propose a second symbolic execution based approach that is more rigorous but much less scalable than the first approach. We have compared the two approaches on small programs and evaluated the first one on Parsec benchmark suite and a large industrial-size benchmark MySQL. Our experiments confirm that the randomized execution based approach is effective and practical.

1 Introduction

Nowadays, widespread multicore hardware has put us at a fundamental turning point in software development. Although we have seen incrementally more programmers writing multithreaded programs in the past decade, the vast majority of applications today are still single-threaded and cannot benefit from the hardware improvement without significant redesign. Applications will need to be well-written concurrent software programs in order to benefit from the advances in multicore processors.

The main reason to develop concurrent programs, which are much more sophisticated than sequential programs, is to enhance the performance of an application. To achieve the performance, developers usually make extra effort to hand tune the programs. One aspect of performance enhancement is data locality because of its significant effect on cache. In order to manage locality, developers need to measure the memory distance of their programs.

The *memory distance* of a reference is a dynamic quantifiable distance in terms of the number of different memory references between two accesses to the same memory location [1]. It is a widely accepted concept in analyzing program cache performance. The speed gap between the processor and memory has resulted in what is known as the memory wall. To overcome this wall and speed up program performance, data locality is an important factor that developers must consider. Memory distance analysis [2–4, 1] is an effective method to measure data locality and predict memory behavior.

Much existing work on memory distance measurement and analysis considers sequential programs only. With the trend towards concurrency, we need to do such measurement on concurrent programs. Unfortunately, adapting existing approaches that were designed for sequential programs is not feasible. Due to the inherent non-deterministic behavior under fixed inputs for concurrent programs, measuring concurrent memory distance is fundamentally different from that of sequential programs.

Table 1: Memory reference of a program execution

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Reference accessed	A	B	C	A	C	C	B	A	C	B	A	C	B	B	A	C
Memory distance	∞	∞	∞	2	1	0	2	2	2	2	2	2	2	0	2	2

Consider the example shown in Table 1. The first row lists the indices of the events in a program execution under input vector v. The second row gives the symbolic memory address being accessed and the third row computes the memory distance. In the following, we use an index as the superscript to differentiate the instances of the same memory addresses in the execution trace. The memory distance of A^1, denoted as $\Delta_v(A^1)$, is ∞ because it is the first appearance of A. For the same reason we have $\Delta_v(B^2) = \Delta_v(C^3) = \infty$. $\Delta_v(A^4) = 2$ because there are two accesses to other memory locations between the current access and the previous access to A. Note that $\Delta_v(B^7) = 2$, because although there are four accesses between B^2 and B^7, three out of the four access visit the same memory location. It can be easily observed that the minimal and maximal memory distances under v are 0 and 2 (not considering ∞), respectively. All the existing memory analysis approaches are in general based on such computation, with minor variants [1].

However, the minimal and maximal memory distances under v may not be 0 and 2 if the program under analysis is concurrent. For example, the trace in Table 1 may be from a concurrent program with two threads as shown in Table 2. That is, the first eight memory accesses are from Thread 1 and the remaining eight are from Thread 2. The execution trace in Table 1 corresponds to the case where Thread 2 starts its execution after Thread 1 completes. However, this is

[1] For example, some approaches may report $\Delta_v(B^7) = 4$ because there are four accesses between B^2 and B^7 regardless same memory locations are accessed.

Table 2: Motivating example

index	1	2	3	4	5	6	7	8
Memory references in thread 1	A	B	C	A	C	C	B	A
Memory references in thread 2	C	B	A	C	B	B	A	C

not the only possibility. Many other interleavings are possible, as illustrated in Table 3.

Table 3: Memory distance results in different interleavings

idx	Reference accessed	Memory distance
1	$\{C_2, B_2, A_2, C_2, B_2, B_2, A_2, C_2, A_1, B_1, C_1, A_1, C_1, C_1, B_1, A_1\}$	$\{\infty, \infty, \infty, 2, 2, 0, 2, 2, 1, 2, 2, 2, 1, 0, 2, 2\}$
2	$\{C_2, B_2, A_2, C_2, A_1, B_1, C_1, A_1, C_1, C_1, B_1, A_1, C_2, B_2, B_2, A_2\}$	$\{\infty, \infty, \infty, 2, 1, 2, 2, 2, 1, 0, 2, 2, 2, 2, 0, 2\}$
3	$\{A_1, B_1, C_1, C_2, B_2, A_2, C_2, A_1, C_1, C_1, B_1, A_1, C_2, B_2, B_2, A_2\}$	$\{\infty, \infty, \infty, 0, 1, 2, 2, 1, 1, 0, 2, 2, 2, 2, 0, 2\}$
4	$\{A_1, C_2, B_2, B_1, C_1, A_2, C_2, A_1, C_1, C_2, B_2, C_1, B_1, A_1, B_2, A_2\}$	$\{\infty, \infty, \infty, 0, 1, 2, 1, 1, 1, 0, 2, 1, 1, 2, 1, 1\}$
5	$\{A_1, C_2, B_1, B_2, C_1, A_2, C_2, A_1, C_1, C_2, C_1, B_2, B_1, B_2, A_1, A_2\}$	$\{\infty, \infty, \infty, 0, 1, 2, 1, 1, 1, 0, 0, 2, 0, 0, 2, 0\}$
6	$\{C_2, B_2, A_1, B_1, A_2, C_2, C_1, A_1, B_2, B_2, C_1, C_1, A_2, C_2, B_1, A_1\}$	$\{\infty, \infty, \infty, 1, 1, 2, 0, 1, 2, 0, 2, 0, 2, 1, 2, 2\}$
7	$\{C_2, A_1, B_2, B_1, A_2, C_1, A_1, C_2, B_2, B_2, C_1, A_2, C_1, C_2, B_1, A_1\}$	$\{\infty, \infty, \infty, 0, 1, 2, 1, 1, 2, 0, 1, 2, 1, 0, 2, 2\}$

This simple example illustrates the challenge in measuring memory distance for concurrent programs. Multiple executions of a concurrent program with the same input might exercise different sequences of synchronization events possibly producing different results each time. To obtain accurate memory distances for a given input, *all execution traces* permissible under that input must be examined. However, in current execution environments a developer has no control over the scheduling of threads. Furthermore, when executing a concurrent program by running it repeatedly on a lightly-loaded machine, the same thread interleaving, with minor variations, tend to be exercised since thread schedulers generally switch among threads at the same program locations. The net effect of these impediments is that only a few interleavings end up being examined. This leads to an incomplete picture of memory distances.

In this paper, we present an approach to measure memory distance of concurrent programs. Given the fact that we cannot possibly explore all the thread interleavings of a concurrent program, our approach introduces randomness in repeated executions. By adapting a method called PCT [5], our approach provides a mathematical guarantee to detect memory distances of given triggering depths. That is, if there exists a memory distance d between memory accesses to m with triggering depth δ_m^d(definition to be given in Section 4), our approach guarantees its detection with probability of $1/(n \times k^{\delta_m^d - 1})$, where n and k are the approximated number of threads and the approximated number of events, respectively, of the given program. We have implemented our method in a tool called DisConPro(Memory Distance measurement of Concurrent Programs with Probabilistic Guarantee).

In order to validate the effectiveness of DisConPro, we propose a more rigorous but much less scalable approach to measure memory distance based on symbolic execution. The second approach utilizes the symbolic execution engine that we developed to exhaustively explore all intra-thread paths and inter-thread interleavings. We name this tool DisConSym (Memory Distance measurement of Concurrent Programs based on Symbolic Execution Guarantee). DisConSym can only handle small programs due to its inherent path explosion. By comparing DisConPro against DisConSym on small programs, we are able to determine if DisConPro covers a similar memory distance spectrum as DisConSym.

The contributions of this paper include the following:

1. To the best of our knowledge, we are the first to propose a feasible approach to measure the memory distance of concurrent programs. Our approach is based on randomized executions and provides probabilistic guarantees.
2. We propose a second approach that is more rigorous but less scalable than the first approach. Although such a symbolic execution based approach can only handle small benchmarks, it allows us to evaluate the effectiveness of the first approach.
3. We have implemented two prototypes DisConPro and DisConSym and conducted experiments on medium-sized Parsec[6] benchmarks and a large industrial size benchmark MySQL with DisConPro.

The rest of the paper is organized as follows. The background knowledge of concurrent program execution is described in Section 2, followed by the explanation of our two approaches in Sections 3 and 4, respectively. The experimental results are given in Section 5. Section 6 discusses the related work. Finally Section 7 concludes the paper.

2 Background: Execution of Concurrent Programs

Figure 1 gives a code snippet of a concurrent program with two threads. Depending on the values of a and b, different branches in the two threads can be observed across executions. Depending on the synchronization and operating system scheduling policies, different interleavings can also be observed. In order to present intra-thread paths and inter-thread interleavings, we use the *generalized interleaving graph*(GIG) [7, 8] to illustrate all possible executions of a concurrent program.

Figure 1 depicts the GIG of the code snippet on its left, where black and blue edges represent an execution step of Threads $T1$ and $T2$, respectively. The dashed lines with the same source (defined as b-PP node) denote a branch within a thread and the solid lines with the same source (defined as i-PP node) denote a context switch between two threads. Note that a node can be both b-PP and i-PP. In order to measure memory distance accurately, all the paths in a GIG must be considered. This is what our symbolic execution based approach, described in Section 7, attempts to accomplish. However, enumerating all possible executions is obviously impractical. Thus, we present a practical approach in Section 4.

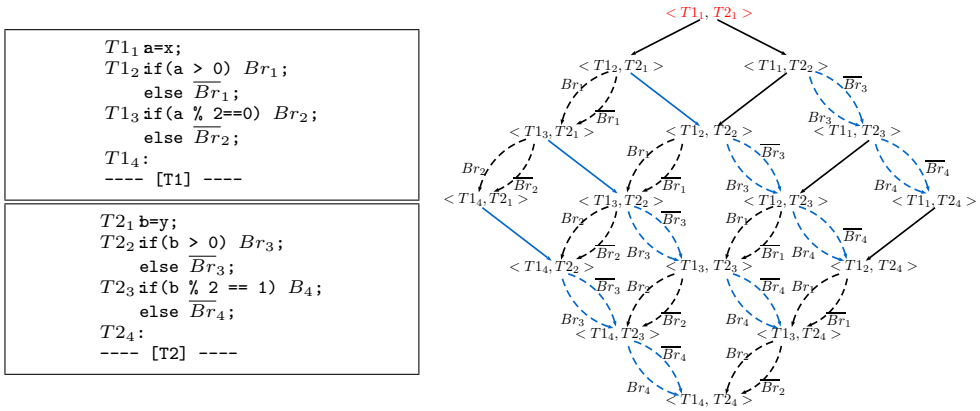


Fig. 1: Code snippet of a concurrent program and its generalized interleaving graph(GIG).

3 Memory Distance Measurement Based on Symbolic Execution

In this section, we present a symbolic execution based approach that is able to systematically explore all the intra-thread branches and inter-thread interleavings. The pseudo-code is shown in Algorithm 1, which is based on the symbolic execution algorithm proposed in [8], and follows the Concolic [9] framework. The algorithm uses a recursive procedure TRACKSTATE to explore paths. The first path is randomly chosen. When a new b-PP node with condition c is encountered, TRACKSTATE checks whether the current path condition appended with c if satisfiable. If so, it continues the execution along the branch while pushing the other branch $\neg c$ on the stack S. The satisfiability is checked by an SMT solver such as Z3 [10]. If the SMT solver fails to find a solution, it indicates that no inputs or interleavings can continue the execution along the branch. In this case, the current execution backtracks by popping its stack S. If an i-PP node is first encountered, TRACKSTATE randomly choose one interleaving while pushing the other one on the stack. For a more detailed explanation, please refer to [8]

The measurement of memory distance occurs during backtrack. That is, when the current execution reaches an end state *normal_end_state* or reaches an infeasible branch. In GETMEMDIST, a path is treated as a sequence of member accesses $\langle acc_1, \ldots, acc_n \rangle$. Each acc_i is a pair $(addr, d)$ of memory address and distance. All the global memory accesses are analyzed to calculate the memory distance. Initially the memory distance of any memory access is set to -1. The algorithm continuously checks the next access acc_j. If acc_j accesses a memory address different from $acc_i.addr$, $acc_j.addr$ is added to the set $memorySet$. Otherwise,the size of $memorySet$ is the memory distance between acc_i and acc_j.

Algorithm 1 SymbolicExecution(P)

```
    let Stack S ← ∅ be the path constraints of a path;
 1: TrackState(s)
 2:     S.push(s);
 3:     if (s is an i-PP node or b-PP node)
 4:         while (∃t ∈ (s.enabled \ s.done \ s.branch))
 5:             s′ ← Next(s, t);
 6:             TrackState(s′);
 7:             s.done ← s.done ∪ {t};
 8:     else if (s is an local thread node)
 9:         t ← s.next;
10:         s′ ← Next(s, t);
11:         TrackState(s′);
12:     path ← S.pop();
13: Next(s, t)
14:     let s be ⟨pcon, M⟩;
15:     if ( t instanceof halt )
16:         s′ ← normal_end_state;
17:         GetMemDist(path);
18:     else if ( t instanceof branch(c) )
19:         if ( s.pcon is unsatisfiable under M )
20:             s′ ← infeasible_state;
21:             GetMemDist(path);
22:         else
23:             s′ ← ⟨pcon ∧ c, M⟩;
24:     else if ( t instanceof X = Y op Z )
25:         s′ ← ⟨pcon, M[X]⟩;
26:     return s′;
27: GetMemDist(path)
28:     let path be ⟨acc_1, ..., acc_n⟩;
29:     for (int i ← 0, i < n − 1, i++ )
30:         memorySet ← ∅;
31:         for (int j ← 1, j < n, j++ )
32:             if (acc_i.addr = acc_i.addr
33:                 acc_i.d ← memorySet.size();
34:                 break;
35:             else
36:                 memorySet.insert(acc_j.addr);
```

4 Memory Distance Measurement with Random Scheduling

In this section, we present our main approach that computes memory distances with random scheduling. We begin with the concept of memory distance minimal depth δ_m^d. Given a memory location m, δ_m^d is defined as the minimal number of constraints for any pair of accesses to m that have a memory distance of d. Consider the example given in Figure 4. There are four threads with eight events

$e_1, \ldots, e_8$ that access four memory locations A, B, C and D. Among the total 2520 interleavings, the memory distances between a pair of accesses to A range from 0 to 3. The memory distance of 3 occurs only if $e_1 \prec e_2 \wedge e_1 \prec e_3 \wedge e_1 \prec e_4$, where $\prec$ denotes the happens-before relation. That is, $\delta_A^3 = 3$ because there are three constraints.

Thread1	Thread2	Thread3	Thread4
$< e1, A >$	$< e2, B >$	$< e3, C >$	$< e4, D >$
$< e5, A >$	$< e6, A >$	$< e7, A >$	$< e8, A >$

Table 4: Four threads with eight memory accesses.

4.1 PCT Algorithm

We adapt the PCT [5] algorithm that was proposed to detect concurrent bugs with a probabilistic guarantee. The basic idea is to add a random scheduling control mechanism to randomize scheduling to avoid redundant executions.

In [5], a concurrent bug depth is defined as the minimum number of order constraints that are sufficient to guarantee to find the bug. The algorithm attempts to find the concurrent bug with depth of d by controlling the thread scheduling as the following.

- The scheduling is controlled by giving each thread a priority. A thread executes only if it has the highest priority or the threads with higher priorities are waiting.
- It assigns n initial priorities d, $d+1$, $d+2$....$d+n-1$ to the n threads.
- It randomly picks $d-1$ change points from k instructions, where k is the estimated number of instructions. The program is then executed with the following rules.
  - Each time only the enabled instruction from the thread with the highest priority can be executed. During execution all the instructions are counted.
  - If the instruction to be executed is counted as the number k-th and k is equal to any of k_i, change the priority value of the current thread to i. This causes a context switch.

4.2 Measure Memory Distance with Random Scheduling

We propose an approach called DisConPro, which adapts the PCT [5] algorithm to measure the memory distance in concurrent programs. As demonstrated above, memory distances may be different with different interleavings.

The basic idea of DisConPro is to measure the memory distance in multiple executions with the PCT scheduling control menchanism. At the begining,

DisConPro generates a random schedule following PCT [5]. Then it executes the program following the schedule. The memory distance is measured during the execution. Each memory access is recorded in a memory access trace and memory distances are calculated based on the memory access traces.

In practice, the statically computed scheduling is not always feasible. However, the infeasible cases only deviate from the planned interleavings but do not lead to execution error. For example, DisConPro may attempt to execute an instruction that is disabled by the operating system. In this case, the execution will choose the next thread with highest priority until an enabled instruction is found.

Algorithm 2 DisConPro(P,n,k,d,m)

Input: P is a program
Input: n is the number of threads
Input: k is the number of events
Input: d is memory distance minimum depth
Input: m is a memory address on which memory distance is measured
1: **Var: Trace** is a list that records every memory access events
2: **Var: Distance** is an array of memory distances
Distance[i] is the memory distance between **i-th** and **(i+1)-th** access to **m**
3: **Trace** = Empty List
4: Generate a random schedule **S** based on PCT algorithm
5: Schedule **n** threads based on **S** and execute those **k** events
6: **for** `each memory access event e` **do**
7: `Trace.add(e)`
8: **end for**
9: Calculate **Distance** based on **Trace**

4.3 Probabilistic Guarantee Inheritance

The PCT algorithm provides a probabilistic guarantee to find a concurrent bug. By adapting it, our approach can provide a probabilistic guarantee to find a particular memory distance d with a depth of δ_m^d. The probability is at least $1/(n \times k^{\delta_m^d - 1})$. Now we now give the proof by adapting the proof for finding a concurrent bug found in [5].

Definition 1. *$DisConPro(m, n, k, P)$ is defined as a set of memory distances of a memory object m. DisConPro finds memory distances during one execution of program P, containing n threads and k instructions.*

Theorem 1 (Probabilistic Guarantee Theorem). *If there exists a memory distance d with a minimum depth memory distance of δ_m^d, the probability of DisConPro finding it in one execution is*

$$Pr(d \in DisConPro(m, n, k, P)) > 1/(n \times k^{\delta_m^d - 1}) \quad (1)$$

Proof. We define an assert statement $assert(m, d)$ as that d is not the memory distance of memory object m in the execution. We define a bug B that can be flagged if the assertion fails. If bug B is detected, d is found as the memory distance of memory m. We define event E_1 as $DisConPro$ finds bug B and event E_2 as DisConPro finds d as the memory distance of m. Base on the definition of E_1 and E_2, we can argue that $E_1 \equiv E_2$. Let $Cons$ be a minimum set of constraints that are sufficient for E_1 to happen. We argue that $Cons$ is one of the minimum set of constraints that are sufficient for E_2 to happen. This bug B is not different from other concurrent bugs hidden in rare schedules. The depth of B equals δ_m^d, which is the size of $Cons$. We defineE_3 as PCT algorithm find B in one execution. Since DisConPro adapts PCT algorithm, we can argue that $Pr(E_2) = Pr(E_3)$. By the definition, we have

$$Pr(E_1 : d \in DisConPro(m, n, k, P)) = Pr(E_2 : DisConPro\ finds\ B) \quad (2)$$

$$Pr(E_2 : DisConPro\ finds\ B) = Pr(E3 : PCT\ finds\ B) \quad (3)$$

It has been proved that(see[5])

$$Pr(E_3 : PCT\ finds\ B) > 1/(n \times k^{\delta_m^d - 1}) \quad (4)$$

Then,

$$Pr(E_1 : d \in DisConPro(m, n, k, P)) > 1/(n \times k^{\delta_m^d - 1}) \quad (5)$$

5 Experiments

5.1 Implementation

We implement DisConPro using PIN [11], a dynamic binary instrumentation(DBI) framework that allows users to insert analysis routines to the original program in binary form. DisConSym is based on Cloud9 [12], a symbolic execution engine built upon LLVM [13, 14] and KLEE [15]. DisConSym has an extension for analyzing concurrent programs since Cloud9 only partially supports concurrency. The extension of Cloud9 follows the algorithm and implemention given in [16]. With the extension, DisConSym can analyze the interleavings not only due to synchronization primitives, which is also supported by Cloud9, but also due to global variables. The latter is essential and a prerequisite to analyze the memory distance of a concurrent program.

5.2 Comparison between DisConPro and DisConSym on small programs

We compare DisConPro with DisConSym to answer the following questions.

- Can DisConPro discover the same memory reuse range as DisConSym does?
- Can DisConPro cover all valid tracks as DisConSym does?
- Is DisConPro more scalable than DisConSym?

Table 5: Impact of the number of global variables comparing with DisConSym and DisConPro

	Thread number = 3	2 *mem_global*	3 *mem_global*	4 *mem_global*	5 *mem_global*
DisConSym	*mem_global* 1	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 2	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 3	N/A	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 4	N/A	N/A	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 5	N/A	N/A	N/A	-1,0,1,2,3,4
DisConPro	*mem_global* 1	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 2	-1,0,1	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 3	N/A	-1,0,1,2	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 4	N/A	N/A	-1,0,1,2,3	-1,0,1,2,3,4
	mem_global 5	N/A	N/A	N/A	-1,0,1,2,3,4

Since DisConSym is not scalable, we compare the two tools on several small concurrent programs with an adjustable number of threads and global variables. All the programs have less than 100 lines of code. Table 5 gives the experimental results. In the experiments we set the number of threads to 3, as indicated by the heading of Column 2, and the number of global variables to be 2-5. DisConSym is not able to handle a program with more threads and global variables. Columns 3-6 indicate the number of global variables created in each group of experiments. Each row in the table gives the memory distance observed for each individual global variable. When a variable does not exist in an experiment, e.g. *mem_global*3 in an experiment with only two global variables in Column 3, N/A is given. In the table, the top half of the rows give the results under DisConSym and the bottom half show the results under DisConPro. For all the experiments done by DisConPro, we set depth to be 5 and run each program 100 times. The table indicates that memory distances can be affected by the number of global variables. It can also be observed that for the small programs DisConPro can find as many memory distances as DisConSym.

Although for small programs DisConSym and DisConPro generate the same results in measuring memory distance, the cost is significantly different. Table 6 gives the number of paths and time usage of the seven groups of experiments with various numbers of threads and global variables. It can be observed that even for such small programs DisConPro is more than 1000 times faster. As concurrent programs become larger, the gap will be wider. Although we cannot guarantee DisConPro can detect as many memory distances as DisConSym does for non-trivial programs, we believe DisConPro achieves a nice trade-off between accuracy and efficiency.

5.3 DisConPro on public benchmarks

We evaluate DisConPro with 9 applications in the Parsec benchmark suite [6], as well as the real-world application MySQL with more than 11 million lines of code. For each application, we conduct 6 groups of experiments. Group one measures the memory distance by only running the test cases once without scheduling

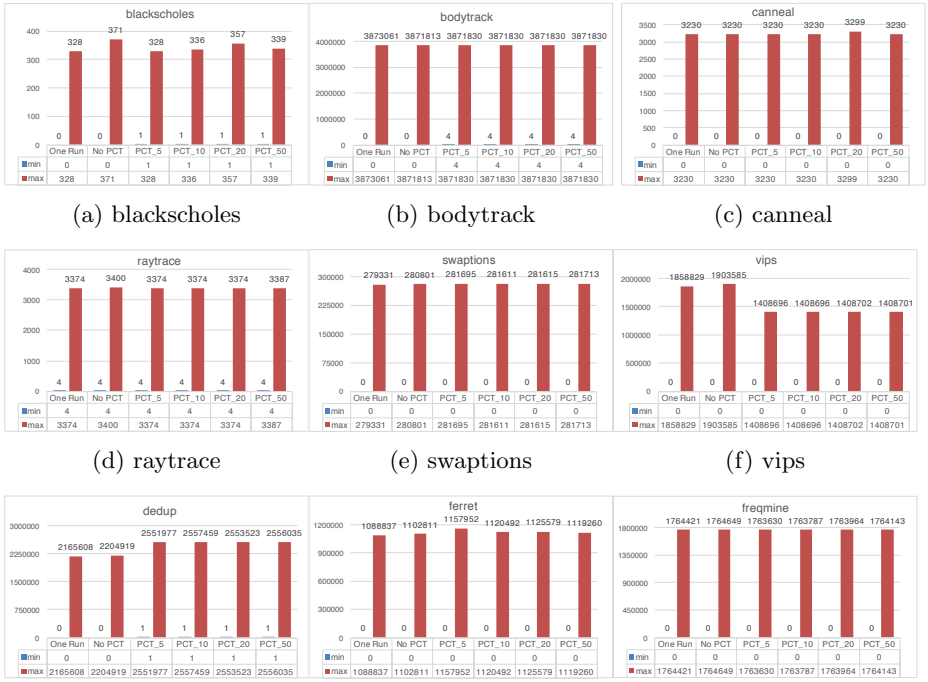
Table 6: Tracked paths and time cost result for DisConSym and DisConPro

Threads and *mem_global* setting	Approach	# Paths	Time (seconds)
3 threads, 2 *mem_global*	DisConSym	90	2
	DisConPro	100	25
3 threads, 3 *mem_global*	DisConSym	1680	27
	DisConPro	100	25
3 threads, 4 *mem_global*	DisConSym	34650	930
	DisConPro	100	25
3 threads, 5 *mem_global*	DisConSym	>200000	>6794
	DisConPro	100	25
2 threads, 3 *mem_global*	DisConSym	20	1
	DisConPro	100	25
4 threads, 3 *mem_global*	DisConSym	>200000	>9609
	DisConPro	100	25
5 threads, 3 *mem_global*	DisConSym	>200000	>11473
	DisConPro	100	25

control. Groups 2 to 6 measure the memory distances by running each test case 30 times. Group 2 uses random scheduling. Groups 3-6 set the predefined depth to 5, 10, 20 and 50, respectively. For each application, we perform memory distance analysis on global variables only. For a large application with too many global variables, we randomly choose several global variables to measure their memory distances.

Parsec Benchmark. Figure 2 gives the results of the experiments on Parsec [6]. The data in the sub-tables and sub-figures present the range of memory distances. Each column gives the minimum and maximum distances of all the global variables we evaluate. The figures show that in most cases the ranges that DisConPro finds are larger than those detected by Random_Schedule, which in turn are larger than the ranges discovered by Single_Run. However, the range gaps achieved by PCT are not comparable to those obtained by random algorithm or even single runs. This is because the ranges reported in the figure are for all the global variables that we have evaluated. Assume that there exists a global variable that is accessed at the beginning of an execution and is re-accessed before the program terminates, its memory distance span is large and does not change much under all the possible interleavings. In this case, this variable hides the differences of the ranges exhibited in other variables.

For the application *vips*, single and random executions without PCT detect a larger memory distance span. Since PCT randomly generates change points to enforce context switches, it may disturb program executions significantly. For this reason, PCT may observe memory distances that are less diverse than those without PCT. This phenomenon is further amplified by the facts that we aggregate all variables in the same figure. To understand the performance of PCT algorithms further, we choose to illustrate the data per variable in MySQL experiments.



(a) blackscholes (b) bodytrack (c) canneal

(d) raytrace (e) swaptions (f) vips

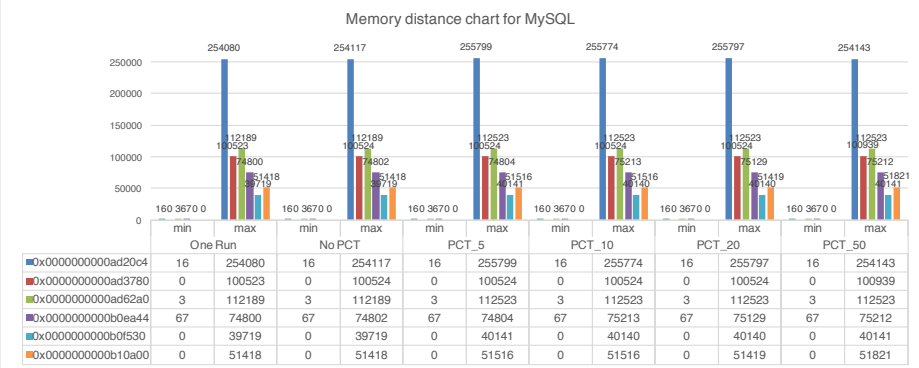
(g) dedup (h) ferret (i) freqmine

Fig. 2: Parsec results

MySQL. Figure 3 gives the experimental results on MySQL. We randomly choose 6 memory objects whose addresses are listed in the table. The figure depicts the ranges of the minimum and maximum memory distances that we have observed from each group of experiments. It can be observed that the memory ranges in Groups 2 to 6 are larger than that in Group 1. By comparing the results of Group 2 to Groups 3-5, we can conclude that DisConPro is more effective than the random scheduling algorithm. The best performance algorithms for the six memory objects are PCT_5, PCT_50, PCT_5 or PCT_10 or PCT_50, PCT_10, PCT_5 or PCT_10, PCT_50, respectively. For measuring the memory distances of individual variables, DisConPro can find a range that is 30% larger than Random_Schedule.

6 Related Work

Cache performance heavily depends on program locality. In the past there were studies that indirectly measure program locality by simulating its execution on a set of probable cache configurations. Such simulations are not only time consuming but also inaccurate. In [17], Ding and Zhong proposed to measure program locality directly by the distance between the reuses of its data because data reuse



	One Run		No PCT		PCT_5		PCT_10		PCT_20		PCT_50	
	min	max	min	max	min	max	min	max	min	max	min	max
0x0000000000ad20c4	16	254080	16	254117	16	255799	16	255774	16	255797	16	254143
0x0000000000ad3780	0	100523	0	100524	0	100524	0	100524	0	100524	0	100939
0x0000000000ad62a0	3	112189	3	112189	3	112523	3	112523	3	112523	3	112523
0x0000000000b0ea44	67	74800	67	74802	67	74804	67	75213	67	75129	67	75212
0x0000000000b0f530	0	39719	0	39719	0	40141	0	40140	0	40140	0	40141
0x0000000000b10a00	0	51418	0	51418	0	51516	0	51516	0	51419	0	51821

Fig. 3: MySQL result

is an inherent program property and does not depend on any cache parameters. They designed two algorithms with one targeting efficiency and the other one targeting accuracy. Their work inspired further improvements that exploits sampling [**?**] and statistical methods [1]. These methods work well for sequential programs. However, they do not consider that non-deterministic thread scheduling and thus not applicable to concurrent programs.

In recent years there has been research on multicore reuse distance analysis [18–21]. Schuff et. al. [18] propose a sampled, parallelized method of measuring reuse distance profiles for multithreaded programs. Whereas previous full reuse distance analysis tracks every reference, sampling analysis randomly selects individual references from the dynamic reference stream and yields a sample for each by tracking unique addresses accessed until the reuse of that address. The sampling analyzer can account for multicore characteristics in much the same way as the full analyzer. The method allows the use of a fast-execution mode when no samples are currently active and allows parallelization to reduce overhead in analysis mode. These techniques result in a system with high accuracy that has comparable performance to the best single-thread reuse distance analysis tools. While our work also conducts reuse distance analysis of multithreaded programs, there exists fundamental difference between their approach and ours. Schuff et al. focus on the hardware while we focus on software. Their goal is to efficiently measure the distance on a more sophisticated multicore. Thus efficiency is a major concern of their research. With the help of the their findings a system designer may design a better cache. We aim to provide a feasible approach that measures the reuse distance of a particular multithreaded program. Therefore non-deterministic thread scheduling is the major concern of our work. With our approach we hope to let programmers understand the behavior of their multithreaded programs regardless of the cache configurations. The two methods are orthogonal and can potentially be integrated. While we strive to diversify the executions of a multithreaded program, the approach proposed in [18] can be used to monitor each execution.

The goal of the approaches in [19–21] are similar to that of [18]. They apply reuse distance analysis to study the scalability of multicore cache hierarchies, with the goal to help architects design better cache systems. In particular, Jiang et. al. [19] introduce the concept of concurrent reuse distance (CRD), a direct extension of the traditional concept of reuse distance with data references by all co-running threads (or jobs) considered. They reveal the special challenges facing the collection and application of CRD on multicore platforms, and present the solutions based on a probabilistic model that connects CRD with the data locality of each individual thread. Wu et. al. [20] present a framework based on concurrent reuse distance and private reuse distance (PRD) profiles for reasoning about the locality impact of core count. They find that interference-based locality degradation is more significant than sharing-based locality degradation. Wu and Yeung [21] extend [20] by using reuse distance analysis to efficiently analyze multicore cache performance for loop-based parallel programs. They provide an in-depth analysis on how CRD and PRD profiles change with core count scaling, and develop techniques to predict CRD and PRD profile scaling. As we mentioned, our focus is to examine program behavior rather than the cache performance. Thus we measure memory distance from a completely different perspective from [19–21].

There exists work that studies reuse distance from other perspectives. Keramidas et al. [22] propose a direct way to predict reuse distance and apply their method to cache optimization. Zhong et al. [23] focus on the effect of input on reuse distance. They propose a statistical, pattern-matching method to predict reuse distance of a program based on executions under limited number of inputs. Shen et al. [24] introduce the time-efficiency model to analyze reuse distance with time distance. Retaining the high accuracy of memory distance, their approach significantly reduces the reuse-distance measurement cost. Niu et al. [25] present the first parallel framework to analyze reuse distance efficiently. They apply a cached size upper bound to restrict a maximum reuse distance to get a faster analysis. Although these approaches are not optimized for multithreaded programs, many of their ideas can potentially be adopted to extend our work.

Our repeated executions of a multithreaded program relies on PCT [5, 26], a randomized algorithm originally designed for concurrent program testing. The advantage of PCT over total randomized algorithms is that PCT provides a probabilistic guarantee to detect bugs in a concurrent program. There has been recent work that adopts PCT for various purposes. For example, Liu et al. [27] introduce a pthread library replacement that applies PCT to support analyzing data races and deadlocks in concurrent programs deterministically. Cai and Yang [28] propose to add a radius to the PCT algorithm so the revised algorithm can efficiently detect deadlocks. However, to the best of our knowledge, we are the first to apply PCT in applications that are not intended to detect concurrency bugs.

7 Conclusion

In this paper, we have presented an approach to measure the memory distance of concurrent programs. Given the fact that we cannot possibly explore all the thread interleavings of a concurrent program, our approach introduces randomness in repeated executions. By adapting the scheduling method PCT, our approach provides a mathematical guarantee to detect memory distances of given triggering depths.

8 Acknowledgements

This work was supported in part by the National Science Foundation (NSF) under grant CSR-1421643.

References

1. Changpeng Fang, S Carr, Soner Onder, and Zhenlin Wang. Instruction based memory distance analysis and its application to optimization. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 27–37. IEEE, 2005.
2. Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 245–257, 2003.
3. Changpeng Fang, Steve Carr, Soner Önder, and Zhenlin Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the 2004 Workshop on Memory System Performance*, MSP '04, pages 60–68, 2004.
4. Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 2–13, 2004.
5. Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.
6. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite:characterization and architectural implications. In *International Conference on Parallel Architecture and Compilation Techniques*, pages 72–81, 2008.
7. Chao Wang, SAID Mahmoud, Aarti Gupta, Vineet Kahlon, and Nishant Sinha. Dynamic test generation for concurrent programs, July 12 2012. US Patent App. 13/348,286.
8. Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 854–865. ACM, 2015.
9. Koushik Sen. Scalable automated methods for dynamic program analysis. Technical report, 2006.

10. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. pages 337–340, 2008.
11. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.
12. Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43(4):5–10, 2009.
13. Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
14. Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.
15. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. pages 209–224, 2008.
16. Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. *ACM SIGPLAN Notices*, 39(11):165–176, 2004.
17. Chen Ding and Yutao Zhong. Reuse distance analysis. *University of Rochester, Rochester, NY*, 2001.
18. Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2010.
19. Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *International Conference on Compiler Construction*, pages 264–282. Springer, 2010.
20. Meng-Ju Wu, Minshu Zhao, and Donald Yeung. Studying multicore processor scaling via reuse distance analysis. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 499–510. ACM, 2013.
21. Meng-Ju Wu and Donald Yeung. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Transactions on Computer Systems (TOCS)*, 31(1):1, 2013.
22. Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 245–250. IEEE, 2007.
23. Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.
24. Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. Locality approximation using time. In *ACM SIGPLAN Notices*, volume 42, pages 55–61. ACM, 2007.
25. Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012.
26. Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ACM Sigplan Notices*, volume 45, pages 167–178. ACM, 2010.

27. Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.
28. Yan Cai and Zijiang Yang. Radius aware probabilistic testing of deadlocks with guarantees. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 356–367, 2016.