Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?

Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen

Computer Science Department
The College of William and Mary, Williamsburg, VA, USA
{jiang,eddy,ktian,xshen}@cs.wm.edu

Abstract. On Chip Multiprocessors (CMP), it is common that multiple cores share certain levels of cache. The sharing increases the contention in cache and memory-to-chip bandwidth, further highlighting the importance of data locality analysis.

As a rigorous and hardware-independent locality metric, reuse distance has served for a variety of locality analysis, program transformations, and performance prediction. However, previous studies have concentrated on sequential programs running on unicore processors. On CMP, accesses by different threads (or jobs) interact in the shared cache. How reuse distance applies to the new architecture remains an open question—particularly, how the interactions in shared cache affect the collection and application of reuse distance, and how reuse-distance–based locality analysis should adapt to such architecture changes.

This paper presents our explorations towards answering those questions. It first introduces the concept of *concurrent reuse distance*, a direct extension of the traditional concept of reuse distance with data references by all co-running threads (or jobs) considered. It then discusses the properties of concurrent reuse distance, revealing the special challenges facing the collection and application of concurrent reuse distance on CMP platforms. Finally, it presents the solutions to those challenges for a class of multithreading applications. The solutions center on a probabilistic model that connects concurrent reuse distance with the data locality of each individual thread. Experiments demonstrate the effectiveness of the proposed techniques in facilitating the uses of concurrent reuse distance for CMP computing.

1 Introduction

Because of the well-known memory wall problem, on traditional architecture, data locality has been one of the most prominent factors that determine the performance of a program. Its importance becomes even more pronounced on modern Chip Multiprocessors (CMP), where cache and memory bandwidth are shared by a growing number of cores.

In decades of locality research on uni-core architecture, two classes of metrics have been used. One is on the hardware level; an example is cache miss rates. The other is on the program level; reuse distance is a representative. Reuse distance is also called LRU stack distance [19], referring to the number of distinct data elements referenced between

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 264–282, 2010.

this and the previous accesses to the same data element [9]. Unlike hardware-level metrics, reuse distance is inherent to a program, independent to hardware configurations but applicable for the performance prediction of various hardware. It is accurate, and from point to point (from one access to another). In contrast, a cache miss rate is an average value over an interval. Furthermore, reuse distance appears to be cross-input predictable for many programs [9,36]. These features make it appealing for a wide range of uses in software refactoring [3], data reorganization [34,36], performance prediction [32,33,17], memory disambiguation [10,11], software-controlled object-level partitioning [16], and so forth.

The rise of multicore has complicated the characterization of data locality. With cache being shared among multiple cores, accesses to memory by a process are not solely determined by that process itself, but also affected by the other processes running on the same chip. The processes (or threads) that co-run on a chip equipped with shared cache are called the *cache sharers* or *co-runners* of one another.

Many recent studies [14,20,21,12] in the architecture area have started to explore the implications of such architectural changes to the application of hardware-level locality metrics. But we are not aware of any such systematic studies on reuse distance.

In this work, we initiate an exploration in that direction. The exploration reveals that in CMP environments, reuse distance loses some of its appealing properties, and that loss impairs many of its uses. However, for a large class of multithreading programs, the loss is remediable through a probabilistic model that connects co-run locality with the memory behaviors of individual cache sharers.

Specifically, our exploration includes three components. First, we analyze the complexities in extending the traditional reuse distance model to CMP environments (*Section 2*). The analysis is based on a straightforward extension of the concept of reuse distance. In the measurement of a reuse distance, the extended concept counts the number of distinct data elements of *all cache sharers* that are accessed between two consecutive references to the same data element. For clarity, we call such a reuse distance *concurrent reuse distance* and the traditional one *standalone reuse distance*. By comparing these two types of reuse distance, we uncover the loss of hardware-independence by concurrent reuse distance and the special challenges in its measurement. We show that the loss of hardware-independence causes a chicken-egg dilemma for performance prediction. Furthermore, the dilemma is hard to resolve through the standard iterative approach.

Second, by drawing on the observations exposed in a recent study, we find that the hardware dependence of concurrent reuse distance can be relaxed for a class of multithreading applications (*Section 3*). Based on the relaxation, we develop a probabilistic model to capture the statistical connections between concurrent reuse distance and standalone data locality for multithreading applications. The model simplifies the attainment of concurrent reuse distance, laying the foundation for many of its uses.

Finally, we evaluate the accuracy of the probabilistic model on both synthetic and real traces (*Section 4*). The results demonstrate that with the probabilistic model, concurrent reuse distance can be obtained in a reasonable accuracy, suggesting its potential for locality enhancement in CMP environments. We conclude the paper with some

discussions on the potential uses and limitations of concurrent reuse distance, some related work, and a short summary.

2 Concept and Properties of Concurrent Reuse Distance

Concurrent reuse distance is a direct extension of the traditional concept of reuse distance (standalone reuse distance). This section discusses the distinctive complexities of concurrent reuse distance and the implications by comparing it with standalone reuse distance. As a preparation, we first review the properties and uses of standalone reuse distance.

2.1 Review of Standalone Reuse Distance and Its Properties

Standalone reuse distance is a widely used locality model on traditional architecture without cache sharing. It is also called LRU stack distance [19], defined as the number of distinct data elements accessed between the current and the previous references to the same element [9]. Its appealing properties include the following.

- *Rigorousness:* Standalone reuse distance is point-to-point, offering a rigorous measurement of locality. In contrast, a cache miss rate is an average value over an interval, and its value depends on the length of the interval.
- *Value:* The value of standalone reuse distance is bounded—no greater than the number of distinct data elements in the program. This property has simplified the search for patterns between standalone reuse distance and program data size [9].
- *Cross-Input Predictability:* A number of studies have shown that the standalone reuse distance histograms of many programs are predictable across program inputs [9,36,24,17]. This property is essential for its uses in program performance prediction.
- *Independence on Hardware:* Standalone reuse distance is a program-level attribute, determined by the program and input data sets, independent to the hardware configurations. But on the other hand, it is strongly related to hardware performance. As shown in Figure 1, from the histogram of reuse distance, it is easy to estimate the cache miss rate of the execution on an arbitrary cache.

Different levels of standalone reuse distance suit different uses. The first class of uses are for cross-architecture prediction of cache miss rates (as illustrated in Figure 1) and program performance [17,33,32]. The used reuse distance histograms are typically on the whole-program level, with the accesses of all data in the execution considered.

The second class of uses is for program refactoring [3], data reorganization [34,36], and software-controlled object-level partitioning [16]. For these uses, the reuse distance is typically on the object level; each reuse distance histogram corresponds to an important data object (e.g., an array) in the program. Such a histogram reflects the match or mismatch of cache and the accesses to the object, offering hints for data transformations or cache partition.

The third class of uses is on the instruction level. From the distance of data store instructions, Fang and others [11] accurately determine on which specific store instruction a load depends, and use that information for memory disambiguation.

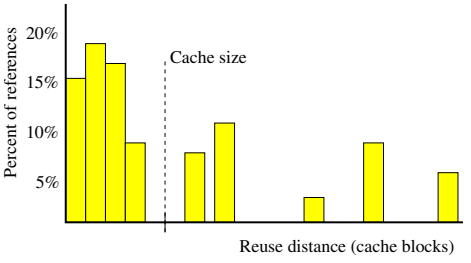

Fig. 1. The histogram of the standalone reuse distance of an execution. Every memory reference on the right side of the cache-size line is considered a cache miss because too many other data have been brought into cache since its previous reference.

2.2 Concurrent Reuse Distance

Concurrent reuse distance is a straightforward extension of standalone reuse distance for programs running on shared cache. It is defined as the number of distinct data elements that *all sharers* of a cache access between the current and the previous references to the same data element. In this section, we consider the general case, where cache sharers can be independent programs, or threads of parallel applications that share the same address space.

Properties. As a straightforward extension, concurrent reuse distance keeps some properties that standalone reuse distance has. It is point-to-point, and its value is bounded, no greater than the sum of the numbers of total distinct data elements of all cache sharers.

However, concurrent reuse distance has a distinctive property:

Its value depends on the relative execution speeds of cache sharers.

Consider two processes, P_1 and P_2, running on a chip with shared cache. The process P_1 conducts a sequence of memory references as $abcba$ (each letter for one data element) during a time interval T (called a *reuse internal*). Without loss of generality, suppose P_1 and P_2 access different sets of data in that interval. The concurrent reuse distance of the second access to a is $2 + x$, where 2 is the number of distinct data elements (b and c) accessed by P_1 in that time interval, and x is the number of distinct data elements accessed by P_2 in that time interval. The value of x depends on the relative speeds of the two processes, $r = Speed(P_2)/Speed(P_1)$. The larger r is, the more data are likely to be accessed by P_2 in that time interval, and hence the greater x tends to be.

Challenges. This property results in some implications important for the measurement and application of concurrent reuse distance.

Challenges to Measurement. Traditional approaches are insufficient for measuring concurrent reuse distance. A typical way to obtain standalone reuse distance is through program instrumentation, which inserts memory monitoring and other relevant instructions

into the program code so that when the program runs, the inserted instructions would collect the memory reference trace and compute the standalone reuse distance.

The instrumented program typically runs hundreds of times slower than the original program does. This slowdown causes inconveniences but no errors to the collection of standalone reuse. However, for concurrent reuse distance, the slowdown would change the relative running speed r among cache sharers, hence causing measurement errors.

To examine the seriousness of this problem, we measure how much the relative speed r changes because of the instrumentation. We use a set of randomly chosen SPEC CPU2000 programs and a dual-core Xeon 7120M with 4MB shared L3 cache. For every pair of the programs, say program i and j, we first run them on two sibling cores and record their respective average IPCs (instructions per cycle), denoted as IPC_i and IPC_j, by reading the hardware performance counters through PAPI [5]. We then use PIN [37] to instrument the programs with the code for the collection of standalone reuse distance. We run the instrumented version on the two sibling cores and record the new IPCs as IPC'_i and IPC'_j. The relative running speeds before and after the instrumentation are $r = IPC_i/IPC_j$, and $r' = IPC'_i/IPC'_j$. We compute the changes of the relative running speed as follows:

$$change\ \ of\ \ relative\ \ speeds = |r - r'|/r. \tag{1}$$

Table 1 reports the results. After instrumentation, the differences of the speeds of those programs become smaller than before. This phenomenon is intuitive considering that the instrumented code dominates the running time of all programs. (The instrumented code is similar for all programs.) The 31–248% changes of the relative speeds caused by the instrumentation suggest the large departure of the measured concurrent distance from the real. This large departure hurts many uses of concurrent reuse distance as most typical uses of reuse distance—such as program refactoring, data reorganization, object-level partitioning, memory disambiguation—have relied on an accurate measurement of the reuse distances.

Deprivation of Hardware-Independence. The reliance on relative execution speeds of cache sharers deprives concurrent reuse distance of hardware-independence. That independence has been a property important for many uses of standalone reuse distance. The deprivation is because the variance in running environments—such as the cache size, the number of cores per chip, the operating systems—often affects the running speeds of different programs in different degrees, and hence changes the relative speeds.

Table 1. Changes of Relative Running Speeds Caused by Program Instrumentation

sharers	1 & 4	2 & 4	3 & 4	1 & 5	2 & 5	3 & 5	1 & 6	2 & 6	3 & 6
r	0.40	0.59	0.25	0.44	0.55	0.43	0.37	0.48	0.30
r'	0.77	0.77	0.87	0.93	0.89	0.93	0.99	1.11	0.88
changes (%)	92.5	30.5	248	111	61.8	116	168	131	193

* programs: 1-ammp; 2-art; 3-mcf; 4-bzip2; 5-gzip; 6-mesa.

Table 2. Changes of Relative Running Speeds Due to Architectural Differences

sharers	1 & 4	2 & 4	3 & 4	1 & 5	2 & 5	3 & 5	1 & 6	2 & 6	3 & 6
r	0.40	0.59	0.25	0.44	0.55	0.43	0.37	0.48	0.30
r'	0.40	0.48	0.3	0.57	0.72	0.54	0.37	0.48	0.31
changes (%)	0	18.6	20.0	29.5	30.9	25.6	0	0	3.3

* r: on Xeon E5310; r': on Xeon 7120M.
* programs: 1-ammp; 2-art; 3-mcf; 4-bzip2; 5-gzip; 6-mesa.

Table 2 shows the changes of the relative speeds when the co-runs happen on a quad-core Intel Xeon E5310 processor with two 4MB L2 cache on each chip, compared to their co-runs on the dual-core Xeon 7120M. Four of the co-runs show negligible changes. Examination shows that the IPC of each of the programs does change considerably in the two architectures. For instance, in the co-runs of *ammp* and *bzip2* (i.e., 1&4) , the IPCs of *ammp* are respectively 0.34 and 0.79 on the two machines. But the IPC of *bzip2* changes proportionally from 0.72 to 1.97. Their relative speeds hence remain the same. However, the relative speeds of the other five co-runs do change substantially, from 19% to 31%, reflecting the hardware-dependence of concurrent reuse distance.

Chicken-Egg Dilemma. As a consequence of the hardware-dependence, one of the main uses of standalone reuse distance—cross-architecture performance prediction [9,36,24,17]—becomes difficult if not impossible for concurrent reuse distance. The difficulty is a chicken-egg dilemma as illustrated in Figure 2. The ultimate target of the performance prediction is the IPC on the new platform, which is supposed to be predicted from the reuse distance collected on a training platform. This prediction is possible for standalone reuse distance because the distance is inherent to the program (and input data sets) and do not change across architecture. However, the prediction becomes difficult for concurrent reuse distance because of its dependence of architecture. To solve this issue, we need first predict the concurrent reuse distance on the new architecture. This prediction however depends on exactly what the concurrent reuse distance is used to predict—the IPC. This inter-dependence forms a chicken-egg dilemma, as showed by the circular flow in Figure 2.

A typical way to solve such kind of dilemmas is through iterative processes. For this particular problem, the process may start with guessing some initial values for the IPCs of co-running jobs. Suppose we have two co-running jobs, I and J and their IPCs are initially guessed to be $IPC_0(I)$ and $IPC_0(J)$. The concurrent reuse distances can then

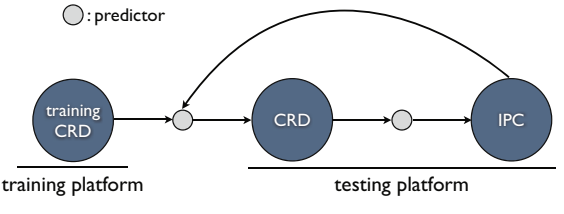

Fig. 2. The difficulty in applying concurrent reuse distance (CRD) to cross-platform performance prediction. The inter-dependence between CRD and IPC forms a chicken-egg dilemma.

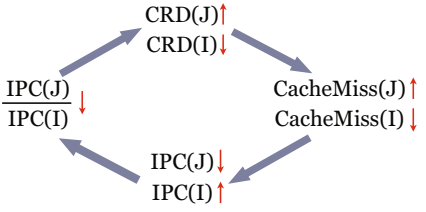

Fig. 3. Analysis showing that the iterative approach cannot solve the chicken-egg dilemma in performance prediction based on concurrent reuse distance

be approximated, denoted as $CRD_0(I)$ and $CRD_0(J)$, from which, the IPCs can be updated accordingly to $IPC_1(I)$ and $IPC_1(J)$. This process continues until reaching a stable point, where the distances or the IPCs remain constant across iterations.

Further analysis however shows that this iterative process does not work for concurrent reuse distance as illustrated in Figure 3. Without loss of generality, assume

$$\frac{IPC_1(J)}{IPC_1(I)} < \frac{IPC_0(J)}{IPC_0(I)}.$$

It means that the relative speed of J (with the speed of I in the respective iteration as the baseline) becomes lower in iteration 1 than in iteration 0. So more data of I would be likely to be accessed in a reuse interval of J in iteration 1 than in iteration 0. The result is that the distances in $CRD_1(J)$ would be larger than those in $CRD_0(J)$. For the opposite reason, the distances in $CRD_1(I)$ would be smaller than those in $CRD_0(I)$. Consequently, the number of cache misses predicted from $CRD_1(J)$ would be higher than from $CRD_0(J)$, leading to a further decrease of $IPC(J)$ and a further increase of $IPC(I)$. Hence, $(IPC_2(J)/IPC_2(I))$ would become even smaller than $(IPC_1(J)/IPC_1(I))$. This decreasing trend would continue for more iterations until the ratio becomes 0 (unless it stops in a local trap).

In summary, this section shows that despite being a direct extension of standalone reuse distance, concurrent reuse distance is both hard to measure and difficult to use in multicore environment. These conclusions are obtained through an analysis of general co-runs. Fortunately, the next section will show that these difficulties can be overcome for a class of important multithreading programs.

3 Concurrent Reuse Distance for Multithreading Programs

The previous section shows the difficulties in applying concurrent reuse distance to *independent* programs co-running on a multicore platform. This section shows that for co-running threads of a *multithreading* program, those obstacles are circumventive.

The solution is based on some recently uncovered features of the execution of multithreading programs on multicore processors. At the kernel of the solution is a probabilistic model that connects concurrent reuse distance with the data locality of each individual thread. We first examine the features of the multithreading programs and then present the probabilistic model.

Table 3. Relative Speeds of Co-Running Threads in Multithreading Applications and the Changes Due to Architecture and Input Variations

input	machine	IPC(thread 0)/IPC(thread 1) for programs						
		blackscholes	bodytrack	canneal	facesim	fluidanimate	streamcluster	swaptions
simlarge	7120M	1.00	0.96	1.00	1.00	1.00	1.00	1.00
	E5310	1.00	1.00	1.00	1.00	0.99	1.00	1.00
native	7120M	1.00	0.92	1.00	1.00	0.99	1.00	1.00
	E5310	1.00	0.99	1.00	1.01	0.99	1.00	1.00
changes by arch. (%)		0	5.9	0	0	0.5	0	0
changes by input (%)		0	2.6	0	0.5	0.5	0	0

3.1 Independence to Architecture and Inputs

A recent study [30] on PARSEC [4], a suite of contemporary multithreading benchmarks, exhibits two phenomena. First, for most of those programs (except pipelining programs), all parallel threads conduct similar computations. Second, the relations among threads, in terms of the amount of shared data and communications, are quite uniform across different thread groups. These phenomena hold across architectures, numbers of threads, assignment of threads to cores, input data sets, and program phases.

The implication to concurrent reuse distance is that contrary to those of the co-runs of independent programs, the relative speeds among threads tend to remain the same across architectures and program inputs. For confirmation, we run all the non-pipelining pthread programs in PARSEC on two types of architectures. One is quad-core Intel Xeon E5310 processors with two 4MB L2 cache on each chip. The other is dual-core Xeon 7120M with 4MB shared L3 cache. We employ two inputs for each program, a small one (*simlarge*) and a large one (*native*). The running times on the two inputs differ by a factor of 43 to 180. In every run, 8 threads are created and are bound to cores such that adjacent threads (e.g., threads 0 and 1) are ensured to run on two sibling cores with cache shared.

Table 3 reports the relative running speeds between two co-running threads (e.g., *IPC(thread 0)/IPC(thread 1)*). The numbers are the average of five repetitive runs (negligible variation appears among the five runs). The bottom two rows are the average changes in the relative speeds— computed in a similar way as Formula 1 in Section 2.2—caused respectively by the variation of architecture and program inputs. The unanimous close-to-1 relative speeds indicate that two co-running threads have virtually the same speeds, no matter on what machine they run or what inputs they use.

We note that the absolute speeds of two co-running threads do change across architectures and inputs. But they change in the same rate so that their relative speed remains the same. As it is the relative speed that matters to the concurrent reuse distance, the results confirm the independence of the concurrent reuse distance of those programs to architecture and input data sets.

3.2 Probabilistic Model for Approximating Concurrent Reuse Distance

The previous section suggests that concurrent reuse distance is potentially useful for a class of multithreading applications. To realize the potential, it is important to explore

the connections between concurrent reuse distance and the memory behaviors of individual threads. The rationale is that if concurrent reuse distance can be derived from the locality information of each individual cache sharer, the appealing properties of standalone locality would directly benefit the prediction and application of concurrent reuse distance.

Overview. We propose a probabilistic model to derive concurrent reuse distance histogram from locality information of each individual thread. The model starts with the locality of individual threads, characterized with *time distance histograms*. *Time distance* is defined as the number of memory references in a reuse interval[1]. In the reference sequence "a b b c a", the time distance of the final access is 4 (while the reuse distance is 2.) *Time distance histogram* is similar to the reuse distance histogram shown in Figure 1 except that the X-axis is replaced by time distance.

The probabilistic model includes two parts. The first computes the number of distinct data elements accessed by each cache sharer in an arbitrary time interval. The second handles the effects that data sharing among threads imposes on concurrent reuse distance. The next two sub-sections explain the two parts respectively.

Part I: From Time to Data Accesses. Let $M^{(j)}(\Delta)$ represent the statistical expectation of the number of distinct data accessed by process j in an arbitrary Δ-long time interval. The goal of this part of the model is to compute $M^{(j)}(\Delta)$ from the time distance histogram of the process j.

The computation includes three steps. *Step 1:* From the time distance histogram of each data object, we calculate the probability for a data object, say O_i, of process j to appear in a Δ-long time interval, denoted by $P_i(\Delta)$. *Step 2:* From $P_i(\Delta)$ ($i = 0, 1, \cdots, N-1$; N is the total number of distinct data objects ever accessed by process j in its entire execution), we obtain the probability for that interval to contain k ($k = 0, 1, \cdots, N$) distinct objects of process j, denoted by $P(k, \Delta)$. *Step 3:* From $P(k, \Delta)$, we compute the expected number of distinct objects that process j accesses in the interval, which is the value of $M^{(j)}(\Delta)$. We explain each of the three steps as follows.

Compute $P_i(\Delta)$

For the object O_i to be accessed in a Δ-long interval, it can be either accessed in the first Δ-1 time points, or, not until the end of the interval. With $q_i(\Delta)$ representing the probability for the data to be not accessed until the end of the interval, $P_i(\Delta)$ can be expressed as

$$P_i(\Delta) = P_i(\Delta - 1) + q_i(\Delta).$$

Hence the following equations:

$$\begin{aligned} P_i(\Delta - 1) &= P_i(\Delta - 2) + q_i(\Delta - 1); \\ P_i(\Delta - 2) &= P_i(\Delta - 3) + q_i(\Delta - 2); \\ \cdots &\quad \cdots \\ P_i(1) &= P_i(0) + q_i(1). \end{aligned}$$

[1] We use logical time—that is, the number of data references—for the length of an interval.

Apparently $P_i(0)$ is 0 (no objects can be accessed in a 0-long interval.) Deduction from these equations produces the following formula:

$$P_i(\Delta) = \sum_{\tau=1}^{\Delta} q_i(\tau). \tag{2}$$

Notice that $q_i(\tau)$ equals the probability for O_i to *1)* be the final data reference in an interval of length τ, and meanwhile, *2)* have a time distance larger than τ at that data reference (otherwise, it would be also accessed at other points in that interval.) With $p_i^{(1)}$ and $p_i^{(2)}$ respectively denoting the probabilities for the two conditions to hold, $q_i(\tau)$ can be computed as $q_i(\tau) = p_i^{(1)} p_i^{(2)}$.

The probability $p_i^{(2)}$ comes directly from the time distance histogram (denoted as H_i) of object O_i as $\sum_{\delta=\tau+1}^{T} H_i(\delta)$. With $p_i^{(1)} = n_i/T$ (n_i is the total references to O_i in all the T data references in the execution), $q_i(\tau)$ can be computed as

$$q_i(\tau) = \frac{n_i}{T} \sum_{\delta=\tau+1}^{T} H_i(\delta). \tag{3}$$

Together, Equations 2 and 3 lead to the following computation of $P_i(\Delta)$ from the time distance histogram:

$$P_i(\Delta) = \frac{n_i}{T} \sum_{\tau=1}^{\Delta} \sum_{\delta=\tau+1}^{T} H_i(\delta). \tag{4}$$

Compute $P(k, \Delta)$ and $M^{(j)}(\Delta)$
With $P_i(\Delta)$ ($i = 0, 1, \cdots, N-1$), we can compute the probability for an interval to contain k distinct data, denoted as $P(k, \Delta)$ as follows:

> $P(k, \Delta) = \sum_S$ (the probability for the interval to contain and only contain all the members of S),

where, S is a k-member subset of $A = \{O_1, O_2, \cdots, O_{N-1}\}$. Using $P_i(\Delta)$, $P(k, \Delta)$ can be computed as follows[2]:

$$P(k, \Delta) = \sum_{S:|S|=k; S \subseteq A} \left(\left(\prod_{i \in S} P_i(\Delta)\right) \left(\prod_{j \in A-S} (1 - P_j(\Delta)) \right) \right). \tag{5}$$

[2] This computation, as most trace-based locality analyses (e.g., [8,25,23]), assumes data distribute independently from one another. Results of those previous studies have shown minor influence of the assumption on locality characterization when the program contains a large number of data.

Recall that $M^{(j)}(\Delta)$ is the statistical expectation of the number of distinct data accessed by process j in an arbitrary time interval of length Δ. According to the definition of statistical expectation, we can compute $M^{(j)}(\Delta)$ from $P(k, \Delta)$ as follows:

$$M^{(j)}(\Delta) = \sum_{k=0}^{min(\Delta-1,N)} k \cdot P(k, \Delta) \tag{6}$$

Discussion. When there are no data sharing among cache sharers, a combination of their $M^{(j)}(\delta)$s ($j = 1, 2, \ldots$,# of sharers) is enough to approximate their concurrent reuse distance histograms. Let d be the time distance of a data reuse by process j. Suppose d_i is the number of memory references by one of its cache sharers, process i, during the same (physical) time period. The concurrent reuse distance of process j can be computed as $M^{(j)}(d) + \sum_{i \in j's\ co-runners} M^{(i)}(d_i)$. (Note, the values of d and d_is may be different, depending on the relative speeds of cache sharers.)

This combination, however, is not sufficient for co-running threads in multithreading applications because of the effects of inter-thread data sharing.

Part II: Handling Data Sharing. In this section, we use the following example for explanation. There are two co-running threads T_1 and T_2. Suppose in a certain time period, the memory reference sequence is

a b X X b X c d X a

where, an X represents some reference conducted by T_2, and the other letters represent the references by T_1. Clearly, this time period corresponds to a reuse interval of reference to "a" in the standalone execution of T_1 with standalone reuse distance of 3 (for accesses to b, c, and d). We now examine its corresponding concurrent reuse distance for element "a" in three scenarios.

- Scenario 1: All Xs are something different from the data accessed by T_1. Let the four Xs be "p q p q". Apparently, the concurrent reuse distance of the reuse interval is just the sum of the numbers of distinct data in each of the two standalone reference sequences: $3 + 2 = 5$.
- Scenario 2: The four Xs are "p a p q". This scenario illustrates the first effect of data sharing. The reference to "a" breaks the reuse interval into two: "a b p a" and "a b p c d q a". The consequence is that the original reuse interval becomes meaningless. The approximation of the ultimate concurrent reuse distances of T_1 has to include a reuse distance of 2 (for "a b p a") and a reuse distance of 5 (for "a b p c d q a").
- Scenario 3: The four Xs are "p c p c". This scenario illustrates the second effect of data sharing. Because "c" is referenced by T_1 in that interval, the references to it by T_2 should not be counted in the concurrent reuse distance. So the resulting concurrent reuse distance is $3 + 1 = 4$ (rather than 5 as in Scenario 1).

The last two scenarios show the two effects of data sharing on concurrent reuse distance approximation.

To approximate the concurrent reuse distance of co-running threads, we first assume no data shared across the threads, and apply the model described in Part I to compute a

concurrent reuse distance histogram, R' for each thread. We then revise R' by considering the two effects of data sharing. The revision tries to find the statistical expectation of the correct concurrent reuse distance for each reuse interval contained in R'.

To explain the revision step, we first introduce some notations. For simplicity, we assume there are only two co-running threads. Let N_1 and N_2 represent the total numbers of distinct data accessed by thread 1 and thread 2 (in their entire execution), S represent the set of data shared by the two threads. Suppose that there is a reuse interval V with ending elements as e accessed by thread 1 and its reuse distance in R' is d' (which needs to be revised in this revision process). Let n_1 and n_2 be the numbers of distinct data among the data accessed respectively by the two threads in V; both can be computed by Equation 6.

Treating the First Effect. The revision step first treats the interval-breaking effect that data sharing may impose to the concurrent reuse distance (the second effect is temporarily ignored). It computes the probability for the reuse interval V to be broken. That event happens only when the following two events both occur. The first is that e is a shared data element; clearly the probability is $|S|/N_1$. The second is that e ever appears in the references by thread 2 in the interval V; as any of the n_2 data elements could be e, the probability is n_2/N_2. So the probability for the reuse interval to be broken is $(|S|/N_1) * (n_2/N_2)$. Because e may appear anywhere in V, assume the broken effect distributes to all sub-intervals of V uniformly. The probability for the resulting reuse intervals to have reuse distance of α ($\alpha = 0, 1, \ldots, d'$) is the same, $(|S|/N_1) * (n_2/N_2)/(d'+1)$. Hence the number of reuse intervals of distance α in R' should increase by $(|S|/N_1) * (n_2/N_2)/(d'+1)$. Meanwhile, because the original reuse interval is broken, the number of reuse intervals of distance d' in R' should decrease by $(|S|/N_1) * (n_2/N_2)$. We use R'' to denote the resulting histogram after this treatment.

Treating the Second Effect. In the treatment to the second effect of data sharing on concurrent reuse distance, each interval is not breakable as the interval-breaking effect has already been considered. For a reuse interval V in R'', let S_1 denote the set of distinct data among all references conducted by thread 1 in that interval, and S_2 for thread 2. In R'', the reuse distance of that interval would be $n_1 + n_2$. In this step, we want to correct this distance value by considering that there may be some overlap between S_1 and S_2. Let C represent the overlap set. Apparently, $C \subseteq S$. The probability for $|C| = c$ is

$$\frac{1}{\binom{N_1}{n_1} * \binom{N_2}{n_2}} \sum_{d=c}^{|S|} \binom{|S|}{d} \binom{N_1 - |S|}{n_1 - d} \binom{d}{c} \binom{N_2 - d}{n_2 - c},$$

where, $\binom{N_1}{n_1} * \binom{N_2}{n_2}$ is the possible ways to have a reuse interval like V, $\binom{|S|}{d}\binom{N_1-|S|}{n_1-d}$ is the number of ways for d shared data to appear in S_1, and $\binom{d}{c}\binom{N_2-d}{n_2-c}$ is the number of ways for thread 2 to access c data in the d shared data accessed by thread 1.

Those probabilities are enough to compute the statistical expectation of the concurrent reuse distance for every reuse distance in R'. Although our explanation uses two threads as the example, the model supports an arbitrary number of co-running threads.

4 Evaluation

This section reports the accuracy of the concurrent reuse distance produced by the probabilistic model. We use both the traces from real programs and some synthetic traces for the evaluation. The synthetic traces allow us to test memory reference patterns that are not covered by the selected programs.

4.1 Synthetic Traces

In order to test the model on traces with various data reuse patterns, we develop a trace generator that produces data reference traces according to users' specifications. The parameters that control the generated trace include the following:

- $n_1, n_2, \ldots, n_k$: the number of unique data blocks (in the unit of cache lines) in the co-running programs.
- s: the data sharing rate. It is the total number of shared data blocks divided by n_1.
- *distribution*: the distribution of standalone reuse distances. We test the following typical distributions: the random, the exponential ($\lambda = -0.97$), the Normal ($mean = 100$, $std. = 33$). Choosing these distributions is because they have been widely used as the primitive distributions in statistical mixture models [13]; the reuse patterns in many real traces can be regarded as the combination of those distributions [23].

The underlying scheme of the trace generator is a stochastic process similar to the one used in standalone reuse distance studies [22].

Table 4 presents the accuracies on a set of traces. The bottom three groups above the average row are the results when there are four co-runners, among which, the first pair both have n_1 unique data items, and the second pair both have n_2.

Following previous work [9], we define accuracy as $(1 - E/2)$, where E is the sum of the absolute differences between the predicted and the real reuse histograms at every reuse distance. Division by 2 normalizes the accuracy to $[0, 100\%]$. To completely expose prediction errors, we use the finest granularity: The width of each bar in all the histograms used in this experiment is 1.

The overall average accuracy is 87.9%. For larger-grained histograms (e.g., 1K-wide bars in many real uses), the accuracy would be higher as errors inside a bar would be smoothed out. The results also show that the effectiveness of the prediction approach is not significantly sensitive to reuse patterns, indicated by the similar accuracy across distributions. The presence of data sharing reduces the prediction accuracy by 5–7%, reflecting the extra complications caused by the sharing to concurrent reuse distance approximation. For most cases, the prediction accuracy is above 80%, verifying the existence of the statistical connections between concurrent reuse distance and the memory behaviors of individual threads, and demonstrating the capability of the probabilistic model in capturing such connections.

4.2 Traces from Real Programs

Because instrumentation changes the relative speeds of cache sharers, the real memory traces of co-running threads are difficult to collect on real machines. For our evaluation purpose, we employ a simulator to record the traces. The simulator is constructed

based on SIMICS [38] with GEMS [18], a cycle-accurate multiprocessor simulator. The simulated system is a dual-core UltraSPARC architecture with 1MB shared L2 cache.

We simulate three representative PARSEC programs [4]. For each program, we use the fast mode of the simulator to move into the region of interest (the labels to those regions come with the original benchmarks) and then collect memory references in one-million-cycle-long detailed simulation.

Program *swaptions* is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The program uses few (23) locks. There are 27% data that are shared between two threads in the collected memory reference trace. The prediction accuracy by the probabilistic model is 74%. The accuracy is relatively lower than those on synthetic traces. The reason is that this program accesses distinct data elements more frequently than the synthetic traces. The reuse distance tends to span a broader range.

Program *vips* is based on the VASARI Image Processing System (VIPS). It includes fundamental image operations such as an affine transformation and a convolution. The program uses locks intensively. There are totally over 33,000 locks. But there are negligible portion of data that are shared between threads. The probabilistic model is able to predict the concurrent reuse distance by 76% accuracy.

The last program is *streamcluster*. It is an RMS kernel developed by Princeton University that solves the online clustering problem. It is a data-level parallel program. This program uses modest number of locks, but many barriers (129,600). There are 3% data

Table 4. Accuracy of the Prediction of Concurrent Reuse Distance Histograms

distr.	s=0		s=10%		s=20%		average
	n_1=200 n_2=100	n_1=200 n_2=200	n_1=200 n_2=100	n_1=200 n_2=200	n_1=200 n_2=100	n_1=200 n_2=200	
random	94.9	93.3	91.3	90.0	89.7	79.8	89.8
expon.	93.2	92.3	91.1	92.2	93.4	90.1	92.1
normal	95.9	94.6	94.4	80.8	93.4	91.6	91.8
random+ expon.	94.0	93.3	88.5	87.2	84.0	79.0	87.7
random+ normal	93.9	93.5	87.4	90.9	91.6	89.1	91.1
expon.+ normal	93.6	94.2	92.5	79.9	92.2	89.9	90.4
2random+ expon.+ normal	88.2	88.5	83.3	82.0	82.5	81.6	84.4
random+ 2expon.+ normal	89.0	84.8	70.1	72.8	85.3	83.5	80.9
random+ expon.+ 2normal	85.0	85.9	84.1	80.0	81.2	81.2	82.9
average	92.0	91.2	87.0	84.0	88.1	85.1	87.9

s: the sharing ratio. n_1, n_2: the number of distinct data of the co-running programs.

shared between two threads in the generated memory reference trace. The approximated concurrent reuse distance histogram has the highest error, 28%. It is mainly due to its irregular data references.

4.3 Discussions

The significance of the model is that it shows the possibility of deriving concurrent reuse distance from the memory behaviors of individual threads, opening the door to many potential uses of concurrent reuse distance. Some of these uses are similar to how standalone reuse distance is applied to sequential programs running on uni-core processors. Examples include cross-architecture performance prediction [32,33,17], software refactoring [3], locality enhancement [34,36,10,11,16]. With the statistical model and the discoveries in Section 3, all these uses become possible for multithreading applications running on CMP.

Some other potential uses of concurrent reuse distance are specific to multithreading applications. An example is thread scheduling [28,30]. It is well known that using hyperthreads may both increase and decrease the performance of applications [15]. From the predicted concurrent reuse distance histograms, one can estimate the cache miss rates of a variety of numbers of threads co-running on a chip. On a CMP processor with hyperthreads enabled (such as Intel Nehalem), that prediction will help determine whether to use hyperthreads or not and how many threads to spawn would yield the best performance.

In our experiments, the longest run of the model takes about 20 seconds. There are many ways to reduce the overhead, such as memory reference sampling [35], employment of coarse-grained histograms, and use of mathematical approximation formulas [23]. Recall that the goal of this work is to reveal the inherent properties of concurrent reuse distance, including its connections with standalone reuse distance—what the probabilistic model captures. Creating a lightweight tool for concurrent reuse distance approximation is orthogonal to the main goal of this work. So sophisticated overhead reduction remains our future work.

5 Related Work

Since the early days in computing [8,19], decades of efforts have contributed a solid foundation for understanding the behavior of dedicated cache systems. Standalone reuse distance has been one of the most influential locality metric [19,2,17,11,36].

However, reuse distance has not been systematically studied in the environment of multicore with cache sharing. The studies close to this work include the following several explorations in predicting miss rates on shared cache.

Ding and Chilimbi [31] have proposed an approach to all-window profiling for concurrent executions, through which, they found that memory accesses by multiple threads of a server application typically show non-uniform interleaving patterns. Chandra et al. [6] have developed three statistical models to predict cache miss rates of co-running processes from the circular stack distance histograms of individual process. Chen and Aamodt [7] extend the models to predict cache contention on Simultaneous Multithreading architecture. Our work differs from these studies in three aspects. First, their

models predict cache miss rates rather than concurrent reuse distance. As a program-level locality characterization, reuse distance has a variety of uses besides performance prediction, such as software refactoring [3], guiding data transformations [34,36], memory disambiguation [10,11]. It is not clear how the previous models apply to these uses. Second, the previous models are for independent jobs, while our model allows data sharing among cache sharers. Finally, because of the use of circular stack distance histograms, the previous models have certain but limited cross-architecture predictive capability. They require that the number of cache sets must remain the same, and the cache associativity of the new machine must be smaller than the cache associativity of the training architecture.

Berg et al. [1] propose a statistical model to estimate the miss rate of shared cache for multithreading programs. Unlike the previously mentioned two studies, their model starts directly from concurrent reuse distance. They assume that the concurrent reuse distance of the interleaved memory reference traces is already available somehow (they obtain it through a simulator), and use it as one of the inputs to their statistical model. The authors collect the traces using simulator for their experiment. It is not clear how such traces can be obtained on real machines. Our model concentrates on the attainment of concurrent reuse distance.

In addition, there has been some work on analyzing the interactions among different threads on dedicated cache in a time-sharing environment [29,27]. These studies mainly focus on predicting the footprint size of a thread as the interactions on cache mainly occur at context switch time; while with shared cache, the interactions happen at almost every cache access—footprint size prediction becomes insufficient.

There have been a wealth of research trying to optimize shared cache performance through either hardware extensions[14,20,21], or operating system scheduling [12,26,28]. They mainly rely on hardware-level locality information collected by hardware performance monitors or special hardware extensions. As mentioned in Section 1, reuse distance differs from hardware-level metrics. It is hardware-independent and captures program-level locality characterizations, important for a variety of locality analysis and program optimizations.

6 Conclusions

The explorations described in this paper lead to the following conclusions. First, despite the wide applicability of reuse distance on traditional architectures, applying it to CMP environments is challenging. The obstacles stem from the reliance of concurrent reuse distance on the relative running speeds among cache sharers. The reliance makes the measurement of reuse distance difficult as instrumentation would change the relative speeds. It also deprives reuse distance of its hardware-independence, impairing many of its uses. Second, experimental evidences show that the relative speeds of many non-pipelining multithreading applications remain unchanged across architectures and inputs because of the uniformity among threads. That observation grants reuse distance the potential applicability for multithreading applications running on CMP environments. Finally, a probabilistic model shows the promise of facilitating the realization of such potential by offering a mechanism to derive concurrent reuse distance histograms from the memory behaviors of individual threads.

Despite the findings and revealed potential, there is no doubt that much further studies are needed before concurrent reuse distance can be practically applied. This work hopefully can help stimulate such studies to systematically extend the commonly used locality model, reuse distance, to modern CMP environments.

Acknowledgments

We owe the anonymous reviewers our gratitude for their helpful comments on the paper. The discussions with Chen Ding's group at University of Rochester helped the refinement of the final version of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations ex- pressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or IBM.

References

1. Berg, E., Hagersten, E.: Fast data-locality profiling of native execution. ACM SIGMETRICS Performance Review 33, 169–180 (2005)
2. Beyls, K., D'Hollander, E.H.: Reuse Distance as a Metric for Cache Behavior. In: Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems (2001)
3. Beyls, K., D'Hollander, E.: Discovery of locality-improving refactoring by reuse path analysis. In: Gerndt, M., Kranzlmüller, D. (eds.) HPCC 2006. LNCS, vol. 4208, pp. 220–229. Springer, Heidelberg (2006)
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Toronto, pp. 72–81 (2008)
5. Browne, S., Deane, C., Ho, G., Mucci, P.: PAPI: A portable interface to hardware performance counters. In: Proceedings of Department of Defense HPCMP Users Group Conference (1999)
6. Chandra, D., Guo, F., Kim, S., Solihin, Y.: Predicting inter-thread cache contention on a chip multi-processor architecture. In: Proceedings of the International Symposium on High Performance Computer Architecture (2005)
7. Chen, X.E., Aamodt, T.M.: A First-Order Fine-Grained Multithreaded Throughput Model. In: Proceedings of the International Symposium on High-Performance Computer Architecture, Raleigh, pp. 329–340 (2009)
8. Denning, P.: Thrashing: Its causes and prevention. In: Proceedings of the AFIPS 1968 Fall Joint Computer Conference (1968)
9. Ding, C., Zhong, Y.: Predicting Whole-Program Locality with Reuse Distance Analysis. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, pp. 245–257 (2003)
10. Fang, C., Carr, S., Onder, S., Wang, Z.: Instruction Based Memory Distance Analysis and its Application to Optimization. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp. 27–37 (2005)
11. Fang, C., Carr, S., Onder, S., Wang, Z.: Feedback-directed Memory Disambiguation Through Store Distance Analysis. In: Proceedings of the 20th ACM International Conference on Supercomputing, Cairns, Queensland, Australia, pp. 278–287 (2006)

12. Fedorova, A., Seltzer, M., Smith, M.D.: Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 25–38 (2007)
13. Hastie, T., Tibshirani, R., Friedman, J.: The elements of statistical learning. Springer, Heidelberg (2001)
14. Hsu, L.R., Reinhardt, S.K., Lyer, R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, Seattle, pp. 13–22 (2006)
15. Liao, C., Liu, Z., Huang, L., Chapman, B.: Evaluating OpenMP on Chip Multithreading Platforms. In: Proceedings of International Workshop on OpenMP (2005)
16. Lu, Q., Lin, J., Ding, X., Zhang, Z., Zhang, X., Sadayappan, P.: Soft-OLP: improving hardware cache performance through software-controlled object-level partitioning. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 246–257 (2009)
17. Marin, G., Mellor-Crummey, J.: Cross architecture performance predictions for scientific applications using parameterized models. In: Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems, New York, pp. 2–13 (2004)
18. Martin, M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News, 92–99 (2005)
19. Mattson, R.L., Gecsei, J., Slutz, D., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM System Journal 9(2), 78–117 (1970)
20. Rafique, N., Lim, W., Thottethodi, M.: Architectural support for operating system-driven CMP cache management. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 2–12 (2006)
21. Settle, A., Kihm, J.L., Janiszewski, A., Connors, D.A.: Architectural Support for Enhanced SMT job scheduling. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 63–73 (2004)
22. Shen, X., Shaw, J.: Scalable Implementation of Efficient Locality Approximation. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 202–216. Springer, Heidelberg (2008)
23. Shen, X., Shaw, J., Meeker, B., Ding, C.: Locality approximation using time. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (2007)
24. Shen, X., Zhong, Y., Ding, C.: Regression-based multi-model prediction of data reuse signature. In: Proceedings of the 4th Annual Symposium of the Las Alamos Computer Science Institute, Sante Fe, New Mexico (2003)
25. Smith, A.J.: On the Effectiveness of Set Associative Page Mapping and Its Applications in Main Memory Management. In: Proceedings of the 2nd International Conference on Software Engineering, pp. 286–292 (1976)
26. Snavely, A., Tullsen, D.M.: Symbiotic jobscheduling for a simultaneous multithreading processor. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 66–76 (2000)
27. Suh, G.E., Devadas, S., Rudolph, L.: Analytical Cache Models with Applications to Cache Partitioning. In: Proceedings of the 15th international conference on Supercomputing, Sorrento, Italy, pp. 1–12 (2001)
28. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. SIGOPS Oper. Syst. Rev. 41(3), 47–58 (2007)
29. Thiebaut, D., Stone, H.S.: Footprints in the Cache. ACM Transactions on Computer Systems 5(4) (1987)
30. Zhang, E.Z., Jiang, Y., Shen, X.: Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2010)

31. Ding, C., Chilimbi, T.: All-Window Profiling of Concurrent Executions. In: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 265–266 (2008)
32. Zhong, Y., Dropsho, S.G., Ding, C.: Miss Rate Prediction Across All Program Inputs. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (2003)
33. Zhong, Y., Dropsho, S.G., Shen, X., Studer, A., Ding, C.: Miss rate prediction across program inputs and cache configurations. IEEE Transactions on Computers 56(3), 328–343 (2007)
34. Zhong, Y., Orlovich, M., Shen, X., Ding, C.: Array Regrouping and Structure Splitting using Whole-Program Reference Affinity. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 255–266 (2004)
35. Zhong, Y., Chang, W.: Sampling-based Program Locality Approximation. In: Proceedings of the International Symposium on Memory Management (2008)
36. Zhong, Y., Shen, X., Ding, C.: Program Locality Analysis Using Reuse Distance. ACM Transactions on Programming Languages and Systems 31(6) (2009)
37. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)
38. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer, 50–58 (2002)