A Highly Efficient and Effective Aggregated Reuse-Distance Approach for Optimal Multi-core System Shared Cache Designs

Hsin-I Wu, Chi-Kang Chen, Hsin-Yu Ho, Chia-Chi Lee and Ren-Song Tsay

Abstract—In this paper, we proposed an effective and efficient multi-core shared-cache design optimization approach based on reuse-distance analysis of the data traces of target applications. Leveraging the fact that data traces are independent of system hardware architectures, our approach allows designers to effectively compute the best cache design at the early system design phase. We devise an efficient and yet accurate method to derive the aggregated reuse-distance histograms of concurrent applications for accurate cache performance analysis and configuration optimization. Essentially, the aggregated reuse-distance histogram information effectively infers the actual shared-cache contention results. The experimental results show that the average error rate of shared-cache performance estimation of our approach is less than 2.4% as compared to actual measured results from real chips. A unique advantage of this approach is that by using a simple scanning search method, one can easily determine the true optimal cache configuration at the early system design phase.

Index Terms—Aggregated reuse-distance, Modeling, Multi-core systems, Multi-level caches

1 Introduction

A good cache design is known to be critical for system performance optimization. However, in the past to optimize design objective of cost and performance, designers need to execute extensive simulations to evaluate designs. Whenever a design configuration is changed, this time-consuming verification process has to be repeated. We propose in this paper an efficient and effective aggregated reuse-distance approach that can dramatically improve system design process, particularly the optimization of cache configurations. The proposed aggregated reuse-distance histograms can be used to estimate the multi-level cache performance of contending concurrent applications accurately. Since the cache miss count calculation is very efficient when using the histograms, we may apply the analysis results to determining the optimal cache designs.

Unlike the single-core systems, certain caches are shared in the multi-core systems. Shown in Fig. 1(a) is a typical 2-level-cache single-core system with a first-level (L_1) cache and a second-level (L_2) cache, which also serves as the last level cache (LLC) before accessing the main memory. Since only one application can take on a core at one time, the running application fully utilizes the entire last level cache space. However, for the two-core system illustrated in Fig. 1(b), each core has a private first-level cache but shares the last level cache. Due to the cache sharing effect, the LLC hit rate of an application concurrently executing on a multi-core system, in general, is worse than the case if the application is solely using the LLC as in the single-core system.

In general, for multi-core systems, different cores may share one cache and execute multiple applications concurrently. With a shared-cache, an application often experiences performance degradation as compared to the case that the application solely uses the cache. The slowdown is mainly contributed by competing for shared-cache usage among applications. Because the resource sharing effect of concurrent applications has greatly increase design complexity, it is critical to understand how the sharing mechanism affects system performance and to provide a systematic early system-level approach for multi-core designs. To ease discussions later in this paper, we adopt a two-level

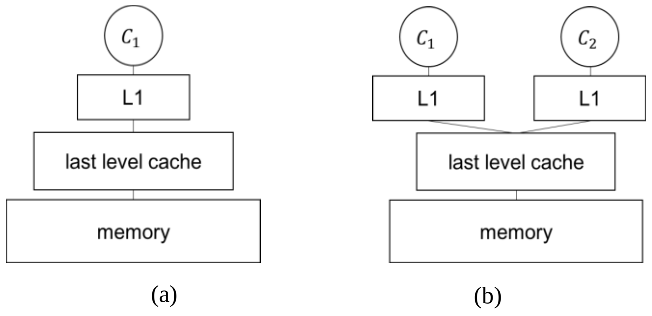


Fig. 1. A typical 2-level-cache processor architecture. (a) A single-core system; (b) A two-core system.

- Hsin-I Wu is with the Computer Science Department, National Tsing Hua University, Hsinchu, Taiwan. E-mail: hiwu.dery@gmail.com
- Chi-Kang Chen is with the Industrial Technology Research Institute, Hsinchu, Taiwan. E-mail: rc@itri.org.tw
- Hsin-Yu Ho is with the Computer Science Department, National Tsing Hua University, Hsinchu, Taiwan. E-mail: shinyu951@gmail.com.
- Chia-Chi Lee is with the Computer Science Department, National Tsing Hua University, Hsinchu, Taiwan. E-mail: jack840709@yahoo.com.tw
- Ren-Song Tsay is with the Computer Science Department, National Tsing Hua University, Hsinchu, Taiwan. E-mail: rstsay@gmail.com

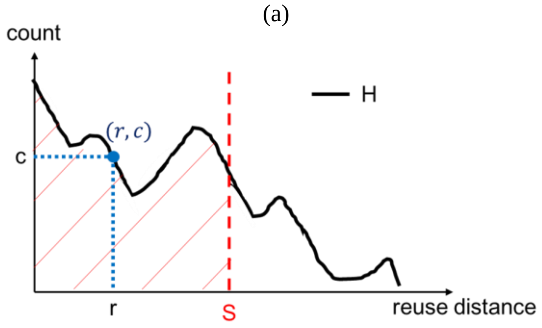
cache design for illustration. However, the conclusion can be generalized to general multi-level cache designs.

To estimate hit/miss rate of a single-core application accurately on a one-level cache, Mattson et al. [16] have proposed using the reuse-distance histogram of the application. Each data access is associated with a reuse-distance number, which is defined as the number of accesses of distinct data memory addresses between current data access and the last access to the same data accessing address. *A* data trace, or a series of memory addresses of continuous data access operations, is illustrated in Fig. 2(a), in which the characters *A, B, C* … symbolize the addresses of data accesses, whereas the number under each access address is the corresponding reuse-distance number. Note that for the first-time occurring data address, the reuse-distance is always marked as ∞ (i.e., infinity), since there is no previous same address data access. Then, in between the 8th memory access address, i.e., *A*, and the last same address access, at the 4th data access, there are two distinct data access addresses, *B* and *D*. Therefore, the reuse-distance of the 8th data access is 2. After collecting the reuse-distance count of each data access, we summarize the number of occurrences of each reuse-distance number and remake a reuse-distance histogram *(H)* as shown in Fig. 2(b), with the reuse-distance number *(r)* on the horizontal axis and occurrence count *(c)* on the vertical axis. For instance, for the memory trace listed in Fig. 2(a), we have the reuse-distance 0 occurs two times, reuse-distance 1 occurs one time, and so on. In other words, a histogram *H* is the collection of all pairs of reuse-distance number and the corresponding occurrence count.

Mattson et al. [16] suggested that for any single-core design with fully associativity cache using LRU replacement policy, one could easily compute the first-level cache hit and miss counts using the reuse-distance histogram. For

Memory trace	A	B	C	A	D	B	B	A	A	B	…
Reuse distance	∞	∞	∞	2	∞	3	0	2	0	1	…

(a)



(b)

Fig. 2. (a) A sample reuse-distance table. (b) An illustrative reuse-distance histogram example - H.

hit count, one simply sums up the occurrence numbers to the left of the reuse-distance number equivalent to the cache size, *S*, as illustrated by the red slashed area and the red vertical line in Fig. 2(b). The corresponding miss count is the sum of the remaining occurrence numbers.

We extended Mattson's method to compute the hit and miss counts accurately on multi-level caches requiring only the first-level reuse-distance histogram, i.e., the one generated from the data trace of the CPU core to the first-level cache. In contrast, Mattson needs to use separate data histogram of each level to calculate hit rate.

Considering cache inclusivity or exclusivity, we accurately calculated hit/miss count of each level cache on the same reuse-distance histogram.

A unique advantage of our approach is that once the target application is determined, one can easily decide data trace and reuse-distance histogram independent of actual cache architecture. In other words, the data property analysis is decoupled from the hardware configurations. Therefore, a designer can easily compute the best cache design using the reuse-distance information of the target applications. Therefore, the method is perfectly suitable for early system-level designs.

Nevertheless, multi-core shared-cache designs are much more complicated as multiple concurrently executing applications interfere with each other and perturb memory traces. For applications executed in a multi-core system, as illustrated in Fig. 1(b), the memory trace consists of memory addresses from all cores in the system. The derived reuse-distance histogram shall be named an aggregated reuse-distance histogram. Intuitively following the single-core case, we can still compute the multi-core hit and miss counts of concurrently executed applications from the aggregated reuse-distance histogram. A designer may use the aggregated reuse-distance histogram to determine an optimal cache size for sharing. However, the issue is that one will need to generate the aggregated reuse-distance histograms of all possible application combinations in advance. It is very time-consuming and impractical to execute all application combinations.

Our major contribution in this paper is that we devise a very efficient and yet accurate method to derive the aggregated reuse-distance histograms from the single-core application reuse-distance histograms for accurate cache performance analysis and optimum cache designs. Our experiments show that our proposed shared-cache hit/miss rate estimation method is highly accurate and efficient, the average error rate is less than 2.4%, and we can effectively design the optimum cache configurations. Details of our approach are to be discussed next.

This paper is organized as the following. We first review related work in Sec. 2 and elaborate the aggregated reuse-distance computation method in Sec. 3. Then, we summarize in Sec. 4 our experimental results and conclude the paper in Sec. 5.

2 Related Work

As a critical subject, many researchers have intensely studied how to improve shared-cache usage for better system throughput. There are two general shared-cache usage approaches: one tries to develop better shared-cache management methods for minimal cache contentions while another attempts to schedule best applications pair for concurrent executions. In general, these approaches all require accurate performance estimation methods based on pre-profiled or runtime information.

To minimize performance degradation

[6][7][8][12][13][14][15], while some proposed specific replacement policies for shared-caches [1][2][3][4][5]. The cache partitioning methods limit each application using a certain part of shared-cache space while the replacement policy approaches try to keep the frequently used data in the cache. These approaches do help to reduce the performance degradation, but a general issue of these approaches is that they all require either additional hardware support or OS modification, which is a non-trivial task.

To avoid needing additional hardware support and OS modification but still be able to predict system performance, some researchers developed cache contention estimation methods based on certain application behavior information [9][10][11][19][20]. For example, Chandra et al. [9] and Xi E. Chen et al. [19][20] profiled reuse-distance statistics (named circular sequences in their paper) of each thread from memory traces and devised probability models for shared-cache miss rate prediction. In general, the probability models are very complex, yet the estimation error rate is close to 10%. In contrast, our proposed aggregated reuse-distance approach is much simpler while providing more accurate shared-cache miss rate calculations.

To avoid processing full memory traces, Eklov et al. [18] proposed a statistical cache sharing model based on certain partial (also called sparse) memory trace of each thread for aggregated reuse-distance estimation. Similarly, Xu et al. [10] and Sandberg et al. [11] used the shared-cache access count derived from the performance counter to estimate the execution performance of each application. In general, these approaches apply a synthetic application that intensively accesses cache segments of various sizes and record the relationship between cache sizes and hit/miss rates. Then with a given cache access count, these methods estimated the effective cache size the target application used and the corresponding hit/miss rate under concurrent executions of other applications. In general, the issue of these approaches is that the partial information may not be representative. Additionally, it is not clear how the cache size or way number may affect the performance estimations. In contrast, our method can reflect the effect of cache size or way number changes.

Instead of requiring pre-profiling, Subramanian et al. [17] performed a runtime method to estimate the performance impact due to shared-cache contentions. Essentially, Subramanian observed that the performance of each application is proportional to the access rate of shared-cache. They compared the measured shared-cache access rate of an application under concurrent execution to the cache access rate when the application is executing alone. The ratio is then used to calculate the performance slowdown of concurrent executions. The issue of this approach is that it requires an additional auxiliary tag store hardware [6] to measure the access rate at runtime. Furthermore, the method is not applicable for early system designs.

In practice, instruction accesses also compete with data accesses and affect performance. Therefore, Jaleel et al. [25] observed that keeping the instruction cache lines in the shared-cache could increase performance, but most other approaches did not consider the effect of the instruction cache. With the proposed aggregated reuse-distance approach, we take the instruction cache into account and can determine the best design configuration at the early system design phase. Moreover, most existing approaches are limited to two-level cache designs whereas our approach effectively handles multi-level cache for multi-core systems.

Details of our proposed approach are elaborated below.

3 Shared-cache Design Optimization

In this section, we first explain how to determine the optimal cache size for a single-core multi-level system by our reuse-distance extension. Then based on the results of a single-core multi-level cache system, we develop the aggregated reuse-distance approach for optimization of the shared-cache designs for multi-core systems.

3.1 Reuse-Distance Extension of Single-core Multi-level Cache

We now discuss how to extend the traditional reuse-distance theory to multi-level cache designs. For the following discussions, we assume that a specific data access β comes with a reuse-distance value R_β. According to the reuse-distance definition, there are R_β counts of unique data requests to the cache in between current and previous accesses of the data β. For convenience, we let S_i be the size of i-th level cache L_i.

We first discuss the case of multi-level exclusive cache, which the higher level cache does not replicate data of lower level cache. For this case, a victim data is pushed to higher level. Therefore, if R_β is greater than S_1, the current access cannot find β in L_1, and it causes a cache miss. In general, if $R_\beta > (\sum_{j=1}^{i} S_j)$ then the data a is missing in L_1,…,

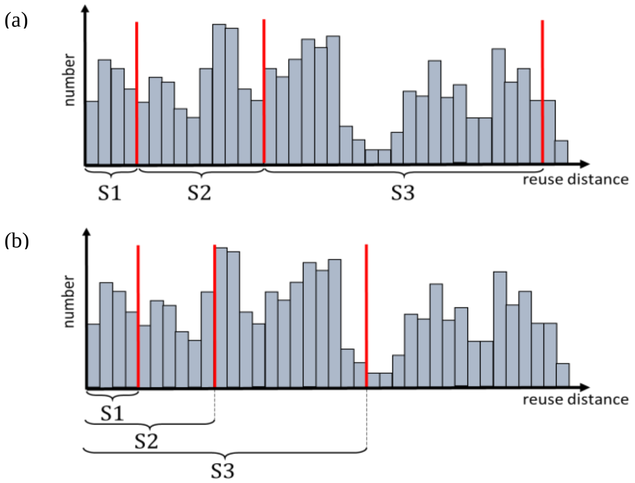


Fig. 3. Compute miss counts for multi-level caches using reuse distance histogram, (a) for exclusive caches (b) and for inclusive caches, where S_i is the size of the i-th level cache.

L_i. Therefore, we may conclude that a miss in Li occurs if the reuse-distance is greater than $(\sum_{j=1}^{i} S_j)$, and hence we can accumulate the counts of those reuse-distances to the right of $(\sum_{j=1}^{i} S_j)$ to get the total count of L_i misses as illustrated in Fig. 3(a).

In contrast, for the multi-level inclusive cache, whose higher level cache always has a data replica of lower level cache, the content of the lower level cache always takes space in the higher level cache. Thus, a data miss in L_i occurs if its reuse-distance is greater than S_i. Therefore, we accumulate the counts of those reuse-distances to the right of S_i to get the total count of L_i misses as shown in Fig. 3(b).

According to the above discussions, we find that we may extend the reuse-distance histograms for evaluating hits/misses of the multi-level cache system. We verify that the so resulted estimation is very accurate as verified by the experimental results in section 4.1. In the following, we further extend the findings and develop a systematic cache optimization method particularly adequate for the early system design phase.

3.2 Scanning Search for Optimal Cache Designs

Equipped with the accurate hit/miss counts estimation method based on the reuse-distance analysis, we now discuss how to find the optimal multi-level cache designs using the scanning search method.

As the CPU microarchitecture is determined, the computation performance is pretty much fixed and what left for system performance optimization is the data access delay through the cache hierarchy to main memory. If the cost is not a concern, an obvious solution for best system performance is to have the fastest memory (or cache) fully deployed. Nevertheless, the cost will be simply impractical.

Therefore, a practical cache design objective is either to minimize the total cache implementation cost c with a target average data access delay value T or to minimize the average data access delay t with the total cache cost constrained by a target value C.

We first discuss the case of minimizing cache cost on one-level cache architecture and later generalize the solution to cover multi-level cases. Suppose that the size of the L_1 cache is x_1; the cache cost function is $c(x_1)$; cache miss rate is $M(x_1)$, and target average data access delay value is T. Also assume that the average execution time per instruction under ideal cache is CPI_{base}, and the miss delay penalty is m, then the cache optimization problem can be formulated as the following:

$$\begin{aligned} min: &\quad c(x_1) \\ s.t. &\quad t = CPI_{base} + m * M(x_1) \le T \end{aligned}$$

Note that $c(x_1)$ shall be a monotonically increasing function and $M(x_1)$ a monotonically decreasing function. Due to the monotonicity of both the cost and miss rate functions, the solution to this problem is in fact quite trivial. We essentially take the smallest cache that can meet the access delay constraint, i.e. $CPI_{base} + m * M(x_1) \le T$. In other word, we have

$$x_1 \le M^{-1}\left(\frac{T - CPI_{base}}{m}\right).$$

Now we discuss how to find the optimal cache size. Although in general, the miss rate function is not a linear function, because the reuse-distance function is not linear, and its inverse function is not trivial, we observe a fact that the cache size is always of the power of 2, i.e., 2^i, in the unit of byte or word. Additionally, since the value 2^i grows exponentially, practically we have limited number of cache size choices. Therefore, an easy solution to finding the optimal cache size is simply scan-searching the few possible choices, such as i=10 to 23 as shown in Fig. 4(a) and the smallest i that satisfies the average data access delay constraint shall be the optimum cache size solution.

We now extend the formulation to handle n-level cache architecture optimization as the following:

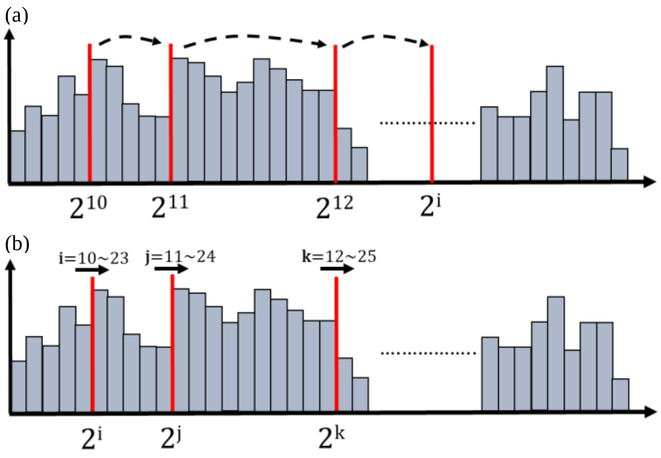


Fig. 4. An illustration of the scanning search method. (a) Scan through the possible size choices and evaluate the corresponding cost for optimal one-level cache design; (b) Scan test on three-level cache combinations.

$$\begin{aligned} min: &\quad c(x_1, \dots, x_n) \\ s.t. &\quad t = CPI_{base} + \sum_{i=1}^{n} m_i * M_i(x_i) \le T \end{aligned}$$

where m_i is the miss delay penalty, and $M_i(x_i)$ is the average miss rate function at level i. Although the cost function $c()$ and the miss rate function $M_i(x_i)$ all preserve the monotonicity property, the solution is not as straightforward as that of the one level cache case. However, as previously discussed the choices of cache at each level are practically of a limited number because the size has to be of a value of the power of 2. Therefore, Fig. 4(b) shows a simple scanning search method which evaluates all possible combinations in the multi-level case. As an example of the three-level cache combinations tested in the experiment, we evaluate cache size choices of the following: i=10~23, j>i, j=11~24, k>j, k=12~25 and the total count of possible cache combinations is merely 560.

If the objective function is the total cache power consumption as modeled in [26][27], then we may have the n-level cache optimization formula

$$\begin{aligned} min: &\quad \sum_{i=1}^{n} [p_{static}^{i}(x_i) + p_{dynamic}^{i}(M_i(x_{i-1}))] \\ s.t. &\quad t = CPI_{base} + \sum_{i=1}^{n} m_i * M_i(x_i) \le T \end{aligned}$$

Note that the total power P consists of two parts. The first one is the static power p_{static} which dissipates energy continuously due to leakage current and is roughly proportional to the cache or memory size. The other is the dynamic power part which is contributed by logic switching caused by data access operations. Therefore, $p_{dynamic}$ is roughly proportional to the miss rate from the last level cache, or the data issuing rate from CPU if it is the first level cache. Obviously, the misses from the last level cache will affect the dynamic power consumption of the main memory. In practice, power consumption model can be generated from tools such as CACTI [26] or detailed circuit simulations.

Again, once the optimization formulation and the cost function are determined, we can easily use the scanning search approach to locate the optimal cache designs based on the reuse-distance as discussed previously.

Similarly, if the object is to minimize the average data access delay subject to a cost constraint, then we can formulate the problem as the following.

$$\begin{aligned} min: &\quad t = CPI_{base} + \sum_{i=1}^{n} m_i * M_i(x_i) \\ s.t. &\quad c(x_1, \dots, x_n) \le C \end{aligned}$$

We then can again apply the scanning search over the reuse-distance to find the best design.

3.3 A Simplified Optimal Single-core Multi-level Cache Design Calculation

Now we assume a simplified ideal uniform step reuse-distance function with a quadratic cost function to derive an optimum analytical solution. Although the assumption is idealistic and may not completely match with practice, the derived result does provide valuable design insights.

We first assume that the reused distance histogram function is a uniform step function as shown in Fig. 5(a), with the maximum reuse-distance cut off value D.

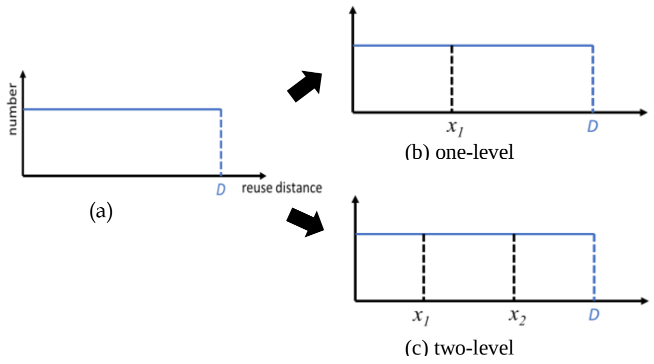


Fig. 5. A simplified analytical model of reuse distance histogram.

For the one-level cache case, the miss rate $M(x_1)$ simply equals to $(D - x_1)/D$. If assume that the size cost function $c(x_1) = ax_1^2$, a quadratic function and a is a positive coefficient. Then, we have

$$\begin{aligned} min: &\quad c(x_1) = ax_1^2 \\ s.t. &\quad CPI_{base} + m * (D - x_1)/D \le T \end{aligned}$$

The above quadratic formulation has an optimum analytical solution, i.e.

$$x_1 = D\left(1 - \frac{T - CPI_{base}}{m}\right)$$

This analytical solution tells us that if the target data access delay T is relaxed to be a larger value, then the cache can be chosen to be smaller. On the other hand, if the instruction computing time CPI_{base} or miss rate m deteriorates to be a larger value, then the cache has to be enlarged to keep the data access delay conform to specification.

Interestingly, if the instruction computation is fast (i.e., small) enough or the miss penalty is small enough, we may not even need cache at all. Equivalently, if the L_1 access latency is too long or the DRAM access latency is very fast, then L_1 is unable to help system performance boost and is simply a waste.

Now for the two-level case as shown in Fig. 5(b), we have

$$\begin{aligned} min: &\quad c(x_1, x_2) = a_1x_1^2 + a_2x_2^2 \\ s.t. &\quad CPI_{base} + m_1\ (D - x_1)/D + m_2\ (D - x_2)/D \le T \end{aligned}$$

Note that the unit cost a_1 of L_1 cache is larger than that of the L_2, i.e., a_2, in practice. Then solving the above formulation, we have the optimum solutions

$$\begin{cases} x_1 = a_2m_1P \\ x_2 = a_1m_2P \end{cases}, \text{ where } P = \frac{D(CPI_{base} + m_1 + m_2 - T)}{a_1m_2^2 + a_2m_1^2}.$$

With the above analytical solutions, we may derive a few interesting design insights. For example, if the unit cost a_2 of L_2 increases, then L_2's size has to become smaller and L_1 larger to balance the delay penalty conflict. If in the opposite, the unit cost a_1 of L_1 increases, we should have a smaller L_1 and a larger L_2. On the other hand, if the miss penalty m_2 of L_2 increases, L_2 should increase in size to reduce access delay while L_1 has to be smaller to abridge cost increase.

The most interesting case is when the miss penalty m_1 of L_1 increases, both L_1 and L_2 have to increase in size rather than just L_1, mainly because $a_1 > a_2$. In this case, the total cost is not minimized if only L_1 takes responsibility for reducing delay increase. Hence, only larger caches of both levels can achieve mia nimal result.

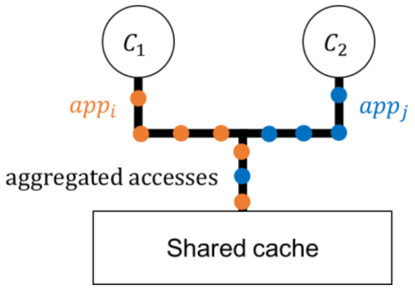
Although the uniform step reuse-distance distribution assumption is somewhat simplistic, the main purpose is to obtain design insights through the analytical solutions and understand how each design parameters may affect the final design cost. In reality, scanning search approach shall provide more realistic solutions.

3.4 Aggregated Reuse-distance of Multi-core Shared-cache

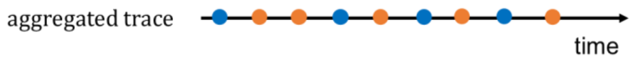
As concluded in section 3.3, designers can now determine the optimal cache size for a single-core multi-level cache system using the reuse-distance extension. Now, based on the reuse-distance extension, we develop aggregated reuse-distance histograms for optimization of the shared-cache designs for multi-core systems. Essentially, an aggregated reuse-distance histogram describes how concurrently executed applications access the shared-cache. Fig. 6 illustrates how the memory traces of two concurrently executing applications app_i and app_j are aggregated into one trace on a two-core system. The aggregated trace is then used to produce an aggregated reuse-distance histogram H' shown in Fig. 7, with the aggregated reuse-distance number r' on the horizontal axis and occurrence count c on the vertical axis, similar to the reuse-distance histogram in Fig. 2(b).

In practice, H' is normally produced from the aggregated reuse-distance histogram Hi' of app_i and Hj' of app_j as shown in Fig. 7. Note that the aggregated reuse-distance for each application has to include other concurrent accesses. Then, for any particular reuse-distance r', if the occurrence counts of app_i and app_j are c_i and c_j respectively, there shall be a corresponding data point (r',c_i+c_j) on H'.

Now with the aggregated reuse-distance histogram H', we may easily determine the optimal cache organization.



(a)



(b)

Fig. 6. (a) An illustration of aggregated memory accesses from two concurrently executed applications. (b) A sample of an aggregated memory trace of the concurrent executed applications.

Following the method of section 3.3, once the cache size of each level is determined, we can effortlessly compute the cache miss rate of each cache hierarchical level from the aggregated reuse-distance histograms. Note that the cache miss rate of each application can be derived from Hi' and Hj'. In other words, with the aggregated reuse-distance histograms, we can precisely analyze the cache sharing effect of concurrently executing applications. However, in practice, there are a few challenges we have to overcome.

First, although the aggregated reuse-distance histogram intuitively can be generated from the aggregated memory access trace of the concurrent executions of target applications as illustrated in Fig. 6, running a laborious simulation of every target application pair is very time consuming and impractical.

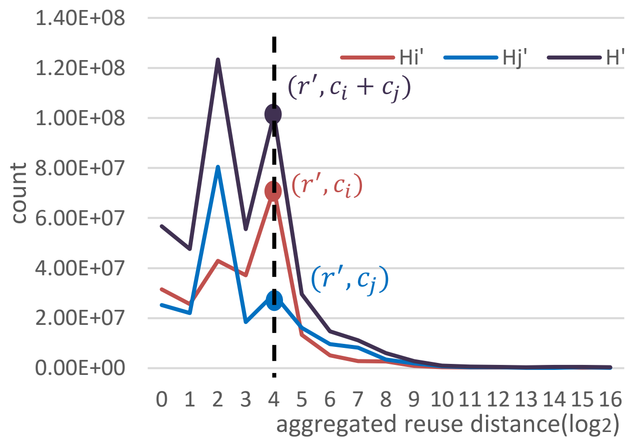


Fig. 7. H' is the aggregated reuse-distance histogram of two concurrent executing applications app_i and app_j. H_i' and H_j' are the corresponding aggregated reuse-distance histograms of app_i and of app_j.

For n target applications, one will need to run simulations of $n(n-1)/2$ application pairs. For example, if 100 target applications are to execute on a two-core system, there are in total 4950 application pairs according to the experimental environment of section 4.2. Each simulation run can take days and therefore have complete simulations is not feasible.

Furthermore, we observe a fact that with different cache size, the same application pair may produce different aggregated traces even if the number of CPU cores is fixed. Since the cache size choice greatly affects cache miss behavior of an application, the merged trace sequence is altered accordingly, although the aggregated traces are in general very similar.

Therefore, to avoid the time-consuming concurrent pair simulations, we devise in the following an effective method that produces accurate aggregated reuse-distance histogram based on the memory trace of each application. The so generated aggregated reuse-distance histograms are verified through thorough experiments to be very accurate for performance analysis and cache design optimization. Details are elaborated in the next section.

3.5 Aggregated Reuse-Distance Computation

We first observe that because of the memory access interleaving effect of concurrent application execution, the reuse-distances of the data accesses of a stand-alone application shall in general increase in numbers in the aggregated memory accesses as illustrated in Fig. 8. Suppose that the reuse-distance of the second access to the address A is r=3 for the stand-alone application app_i. The white dots in the figure represent repeatedly accessed addresses. With the memory access interleavings of a concurrent application app_j, the reuse-distance of the same second access to A shown in the aggregated trace of the concurrent execution is then increased to r'=5 as illustrated.

Specifically, we observe that although there are five

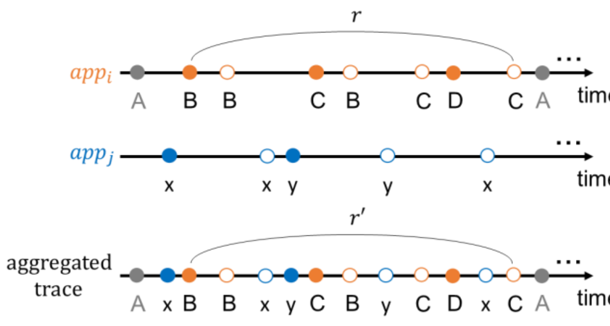


Fig. 8. In the aggregated access sequence, the reuse-distance of a given memory access is to increase to a larger number due to the access interleavings from the concurrent execution of other applications.

data accesses from app_j interleave in between the first and second accesses to the address A in the aggregated trace; there are only two new unique accesses, i.e., x and y to be included in the new aggregated reuse-distance number.

With the above observation, we propose an accurate method to calculate how the original reuse-distance r is to change to r' in the aggregated access sequence. Essentially, we first compute how many data accesses from other concurrent applications are to come in between the current data access and the last access to the same address, and among these two accesses how many are unique accesses (accesses to unique addresses). If the original reuse-distance is r and the number of the new unique accesses is u_r, then the aggregated reuse-distance is $r'=r+u_r$.

In other words, if we can compute the number of unique ones among the interleaving accesses, we can compute the aggregated reuse-distance r' without the need to generate the aggregated trace.

We now discuss how to estimate the number of data accesses that may interleave from other concurrent applications and how to calculate the number of unique accesses among them so that we can compute the increase of the reuse-distance count.

Although without an aggregated trace we cannot directly compute the number of interleavings from other concurrent applications, we may compute the average interleaving accesses between any two consecutive data accesses of the application of concern. Assume that the application of concern is app_i and the concurrent application is app_j. Also, the numbers of data accesses of app_i and app_j are n_i and n_j respectively for a fixed period of execution. Then the average number of app_j data-access interleavings between two consecutive data accesses of app_i is n_j/n_i, which is named the access ratio of app_j to app_i and can be used to estimate the number of interleavings from app_j to app_i.

We illustrate in Fig. 9 the aggregated reuse-distance computation flow, in which the orange dots and blue dots represent the data access from app_i and app_j respectively. Since only the data trace (and reuse-distance histogram) of each application is available, but not the aggregated trace, we compute from the data trace not only the reuse distance number but also the average data-access number of the accesses correspond to each reuse-distance number of the application. Therefore, for each application app_i, we associate each reuse-distance r_i an average data access number $d_{i,r}$ ($\geq r_i$) and construct an *r-d* table as shown in Fig. 9(a).

Then, for each reuse-distance number r_i of app_i, we can look up the corresponding average data access number $d_{i,r}$ from the *r-d* table as illustrated in Fig. 9(b).

Once we have the average data access number $d_{i,r}$, we use it to compute the average reuse-distance increase $u_{i,j,r}$ contributed by the data interleavings from app_j. Note that for $d_{i,r}$ data accesses, there are $(d_{i,r}+1)$ possible interleaving spots as illustrated in Fig. 9(b). Consequently, in average there are $\Delta_{i,j,r}=(d_{i,r}+1)^* n_j/n_i$ interleaved data accesses from app_j as shown in Fig. 9(c), since each interleaving spot can have n_j/n_i interleaved data accesses.

Supposedly, different applications access disjoint data sets. Therefore, the remaining task is to translate the increased number of the interleaved data accesses, $\Delta_{i,j,r}$, to an equivalent increased number of reuse-distance count $u_{i,j,r}$. The question is the same as asking how many unique-address data accesses are there for $\Delta_{i,j,r}$ data accesses in app_j. In fact, this is an inverse calculation as that of computing the data access numbers from the reuse-distance numbers.

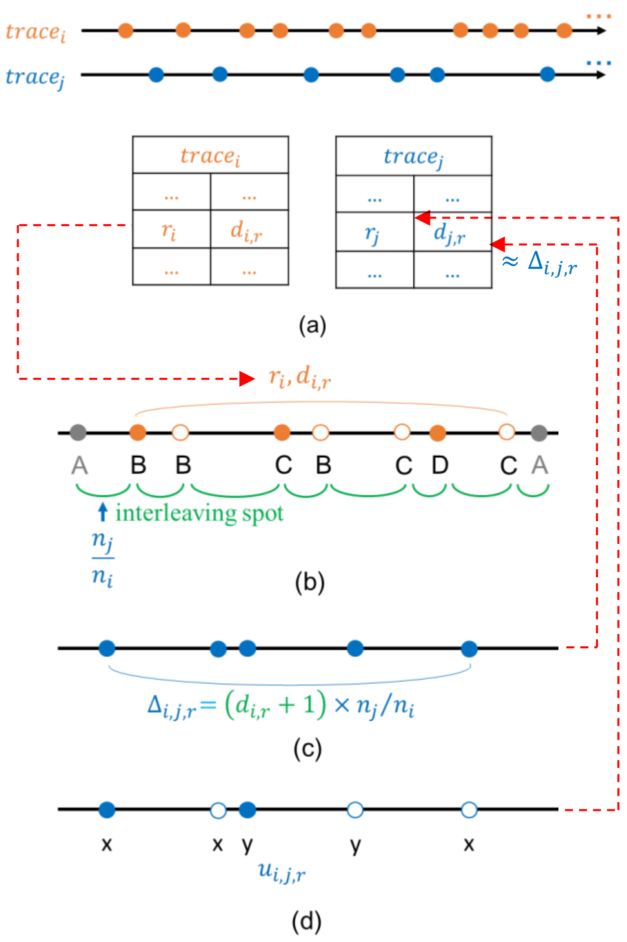


Fig. 9. An illustration of the aggregated reuse-distance computation flow. (a) The *r-d* tables of application traces. (b) Find the average data-access number $d_{i,r}$ for each reuse-distance number r_i from the r-d table; (c): Use the access ratio to compute the interleaving data accesses number $\Delta_{i,j,r}$ from app_j; (d) Use the r-d table to compute the unique access number $u_{i,j,r}$ from $\Delta_{i,j,r}$.

With the average access number $\Delta_{i,j,r}$ from app_j, we find the $d_{j,r}$ which is closet to $\Delta_{i,j,r}$ from the *r-d* table of app_j. Then we use the corresponding reuse-distance numbers r_j to represent the unique access number $u_{i,j,r}$, as illustrated in Fig. 9(d).

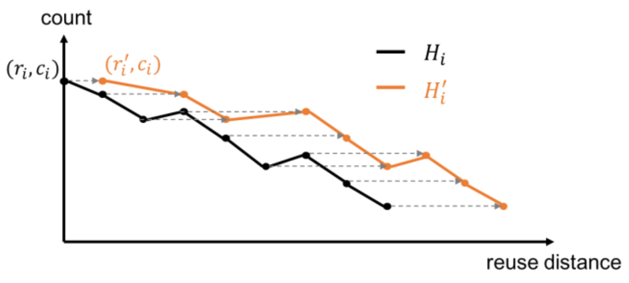


Fig. 10. The aggregated reuse-distance histogram of app_i affected by app_j can be produced by shifting each original reuse-distance point (r_i, c_i) horizontally to ($r_i'= r_i + u_{i,j,r}$, c_i).

Then with this calculated increased reuse-distance number $u_{i,j,r}$, we shift, in the reuse-distance histogram H_i of app_i, the original reuse-distance point (r_i, c_i) horizontally to $(r'_i= r_i + u_{i,j,r}, c_i)$ and have a new aggregated histogram H_i', as illustrated in Fig. 10. Similarly, we can compute the aggregated histogram H_j' from the reuse-distance histogram H_j.

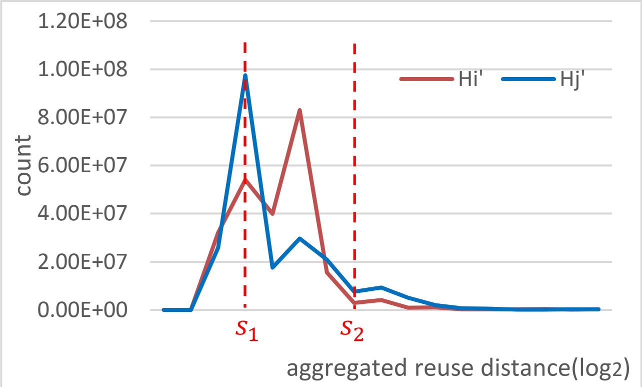


Fig. 11. Given a cache size, one can compute the cache miss rate using the aggregated reuse-distance histograms of the concurrently executing applications

With the aggregated reuse-distance histograms H_i' and H_j' shown in Fig. 11, then according to section 3.3, one can accurately compute the shared-cache hit and miss counts of app_i and app_j on multi-level caches. This method is verified to be very accurate by experiments as discussed in Section 4.

3.6 Optimal Multi-core Shared-cache Designs

The proposed aggregated reuse-distance approach is perfect for determining optimal cache configurations at the early-system design phase. Similar to the single-core case, one possible objective cost function f for the n-level cache is the total performance slowdown due to cache misses, i.e.

$$f(s_1, \dots, s_n) = \sum_{l=1}^{n} m_l[M_{l,i}(s_l) + M_{l,j}(s_l)],$$

where m_l is the delay penalty of the l-th level cache miss, $M_{l,k}(s)$ is the miss count of app_k on the l-th level cache of size s. To calculate the miss count $M_{l,k}(s)$, following Mattson's method [16] one simply sums up the occurrence numbers to the right of (including) the reuse-distance number equivalent to the cache size s on the derived aggregated reuse-distance histograms H_k' as illustrated in Fig. 11.

If we have $m_n=1$ and $m_{l\neq n}=0$, then the objective function

$$f = M_{n,i}(s_n) + M_{n,j}(s_n)$$

represents the total miss count of the last-level cache. Note that the shared-cache partitioning method from Suh [15] mainly try to minimize the total last-level cache miss count with given cache sizes.

Note that in the early system design phase, cache sizes are to be determined. Normally we shall have a total cache cost limit; otherwise, unlimited sized caches always give the minimum miss count. Assume that the total cache cost function is $c(s_1, \dots, s_n)$, where s_l is the size of the l-th level cache.

Then a formal n-level cache design optimization problem can be formulated as the following,

$$\begin{aligned} min: &\quad f(s_1, \dots, s_n) = \textstyle\sum_{l=1}^{n} m_l[M_{l,i}(s_l) + M_{l,j}(s_l)] \\ s.t. &\quad c(s_1, \dots, s_n) \le C, \end{aligned}$$

where C is a designer specified total cache cost limit.

Another possible formulation is to minimize total cache cost under a maximum slowdown limit as the following,

$$\begin{aligned} min: &\quad c(s_1, \dots, s_n) \\ s.t. &\quad f(s_1, \dots, s_n) = \textstyle\sum_{l=1}^{n} m_l[M_{l,i}(s_l) + M_{l,j}(s_l)] \le F, \end{aligned}$$

where F is a designer specified slowdown limit.

To find the optimal solution, we simply adopt the scanning search approach described in section 3.3 to find the optimal solution. Essentially, we evaluate the cost function of each possible cache size used in practice and find the corresponding optimal cache configuration that produces the minimal cost within constraints.

In summary, our proposed approach is very flexible and can be applied to evaluate any designer specified cost and constraint functions. Particularly, the approach is effective for early multi-core system shared-cache designs. Next, we show experimental results to support the effectiveness of our proposed approach.

4 Experimental Results

To verify our proposed optimization method, we adopt SPEC CPU2006 [17] benchmark suite as the target applications and adapt the cache model of SimpleScalar [16] to support three-level cache architecture, both inclusive and exclusive models, for hit/miss evaluations. All experiments are performed on a host machine equipped with Intel Xeon-E5-2650 2.00GHz.

4.1 Single-Core Multi-level Cache Results

Before testing multi-core cache designs, we first verify the results of single-core cache designs. For the experiments, we first use SimPoint [29] to find a representative execution region in each benchmark and then apply Pin [30], a dynamic instrumentation tool, to collect traces of one billion memory accesses of the representative region. Then, we apply the traces to evaluate hit/miss rate of different multi-level cache architectures of various cache size combinations.

For a fair comparison, we first execute the trace-driven

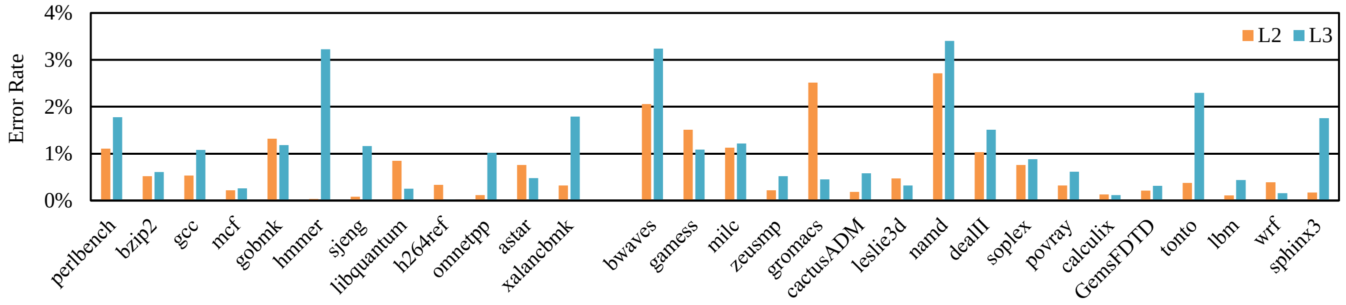


Fig. 12. The average error rate miss count estimation of each SPEC benchamek case on a total 560 cache configurations.

simulation on SimpleScalar and produce golden results for reference. On the other hand, we adopt the parallel reuse-distance histogram computing method [28] to leverage the multi-core host machine to accelerate the computing process. Then we apply our proposed miss rate estimation method discussed in section 3.3. Finally, we compare our estimated result with the golden simulation result and find the error rate to verify the accuracy of the proposed approach.

Regarding the cache size configurations, we list in Table 1 the cache size range of each level. Thus, we compute for each case the 560 possible 3-level cache combinations. Note that the LRU policy is adopted for all tests and the default cache line size of each level is 64 bytes.

Table 1 Size Range for Respective Cache Level

Same Setting for both Inclusive/Exclusive		
L_1	2^i *bytes* , $10 \le i \le 23$	$i, j, k \in \mathbb{N}$, $i < j < k$
L_2	2^j *bytes* , $11 \le j \le 24$	
L_3	2^k *bytes* , $12 \le k \le 25$	

For exclusive cache architecture, our reuse-distance-based method produces the same results as those from the golden simulations. The main reason is that victim data pushed into the next level always resets its LRU value to 0 and hence the relative LRU value is equivalent to that as if we treat both L_1 and L_2 as a union cache. Therefore, the reuse-distance histogram can provide 100% accurate hit/miss rate evaluations for multi-level exclusive cache designs.

In contrast, for inclusive cache designs, the victim data is replaced without affecting the replica in its next level cache, and hence its LRU value is outdated. As a result, it causes slight inaccuracy. In Fig. 12, the average miss error rate of L_2 and L_3 is less than 1%, and the max error is less than 4.46% and 4.78% respectively for all.

The experimental results support that our proposed method is very accurate for evaluating multi-level cache hit/miss counts. As a result, designers may effectively use the proposed hit/miss rate estimation results for optimal cache designs.

We also try to modify the LRU mechanism of inclusive cache simulator, such that the LRU value of the replica data in next level will be updated when the victim data is pushed back to next level. We then confirm that the simulation results are equal to our calculation from the reuse-distance histograms.

Moreover, we show in Fig. 13 the speedup of the total computation time required for 560 simulations and that for reuse-distance histogram calculation. The simulation-based approach takes 150 to 250 times longer computation time than our proposed method. In average, it requires av-

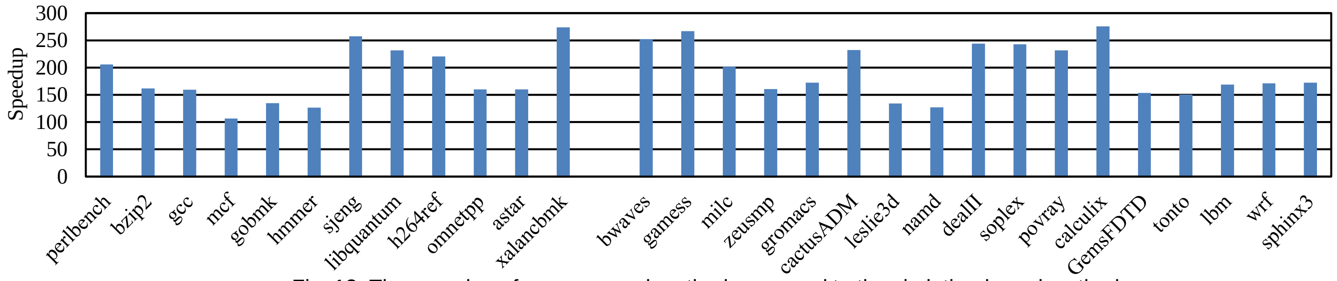


Fig. 13. The speedup of our proposed method compared to the simlation-based method.

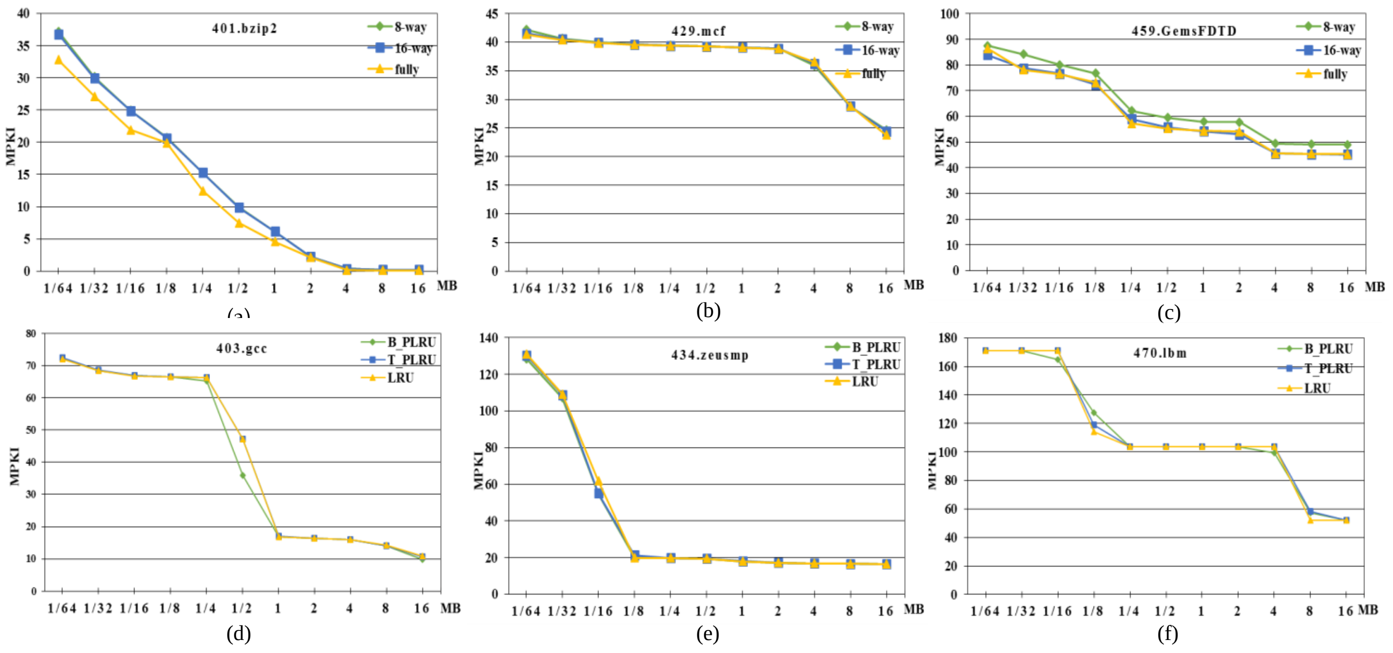


Fig. 14. Comparison of MPKI of N-way with fully-associative designs (a, b, c), and PLRU with LRU replacement policies in 16-ways (d, e, f).

erage only 14 minutes to generate a reuse-distance histogram and estimate the miss rates. Therefore, the reuse-distance extension approach is practical for cache architecture optimization at the early system design stage.

Since in practice and particularly in recent years, Intel and AMD ubiquitously adopt 8-way or 16-way cache designs and bit-based/tree-based Pseudo-LRU replacement policy implementation, we also conduct experiments to verify whether the proposed approach works well for these design practices.

For verification, we compare the MPKI (Misses per Kilo Instructions) of each benchmark under two test settings. The first is to compare the 8-way set, 16-way set, with the fully associative cache all based on LRU. The second one is to compare the bit-based PLRU, tree-based PLRU, with the LRU in 8-way, 16-way, and fully associative caches; Note that we here show only the 16-way's results because the others exhibit similar results. In Fig. 14, we observe only little disparity, and the result supports that the proposed reuse-distance cache optimization approach is effective in practice.

Since increasing cache block size generally may reduce miss rate due to the spatial locality, we further compare the histograms and results produced from different block sizes of the same case. We find that the maximum number of the reuse-distance values is halved for each doubled block size because the total data block number is halved, as illustrated by the example shown in Fig. 15(a). Nevertheless, we may scale the histograms with normalized reuse distance of 1-byte unit as shown in Fig. 15(b) and then apply the scanning search on the three histograms concurrently to determine which block-sized configuration can provide the best result.

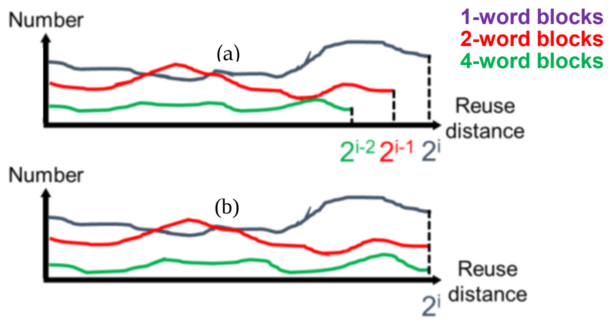


Fig. 15. (a) The original histograms of different block sizes. (b) The normalized historgrams of 1-byte unit.

4.2 Multi-core One-level Shared-cache Results

We follow Jaleel et al. [24] and take the same ten representative test cases, from SPEC CPU2006 [23]suite, listed in Table 2 to test our proposed approach. The ten representative test cases include three high-CPI memory-bound cases, four low-CPI memory-bound cases, and three CPU-bound cases. In fact, there are in total 29 SPEC CPU2006 test cases and 406 pairs of applications. With ten representative test cases, we only need to verify 45 pairs of applications and save 10x testing time while still having solid, unbiased results.

The multi-core simulator Sniper [22] is used to compute the actual miss rates, cache, and execution performance for comparison and validate the estimated results of our approach. In general, we execute the given pair of applications on Sniper and have the faster application executes 400 million instructions. Along with the execution, we record the data access trace of each application and compute the cache miss counts for later references simultaneously.

Table 2: Benchmarks tested from SPEC CPU2006.

Benchmark type	Benchmarks
Memory bound(CPI=4-8)	omnetpp, soplex, lbm
Memory bound(CPI=2-4)	gobmk, hmmer, sphinx3, xalancbmk
CPU bound	dealII, h264ref, sjeng

Before we test on more realistic multi-level cache designs, we first conduct testing on an in-order two-core one-level cache architecture, loading instructions directly from memory and having two cores share the one-level data cache. The shared-cache is fully associative with LRU replacement policy. The purpose of this test is to validate the accuracy of the proposed aggregated reuse-distance histogram generation method focusing on the data contention effect without further mixing of both instruction and data sets.

We use the miss counts computed from Sniper simulations as references and calculate the error rate ε as the following,

$$\varepsilon = \frac{m_s - m_r}{m_s},$$

where m_s is the cache miss count computed from Sniper simulation results and m_r is the miss count estimated by our proposed aggregated reuse-distance computation model. We test ten applications listed in Table 2, which includes 45 application pairs. The experiments show that the geometric mean of the error rates of various cache sizes (512KB, 1MB, and 2MB) is less than 1.8%, which supports the accuracy of our approach. Fig. 16 shows the detailed results.

4.3 Multi-core Two-level Shared-cache Results

To be practical, we also conduct experiments on in-order two-core, inclusive two-level-cache designs with 64-Byte line sized caches. Each core owns a private L_1 instruction and a data cache but shares an L_2 cache, which is the last-level cache to the main memory. The access latencies of the L_1, L_2 caches, and the main memory are one cycle, ten cycles, and 130 cycles respectively. We also use the same 45 application pairs as the test cases.

All experiments are conducted on 8-way and 16-way set-associative caches to be practical. The results show that the reuse-distance-based estimation method works very well for practical cases although our proposed method assumes fully associative, LRU-replacement-policy cache model.

Table 3: Combinations of different cache size and way-associativity form various 2-level cache configurations.

Cache size	16KB_512KB	32KB_1MB	32KB_2MB
Way-associativity	4_8	4_16	F_F

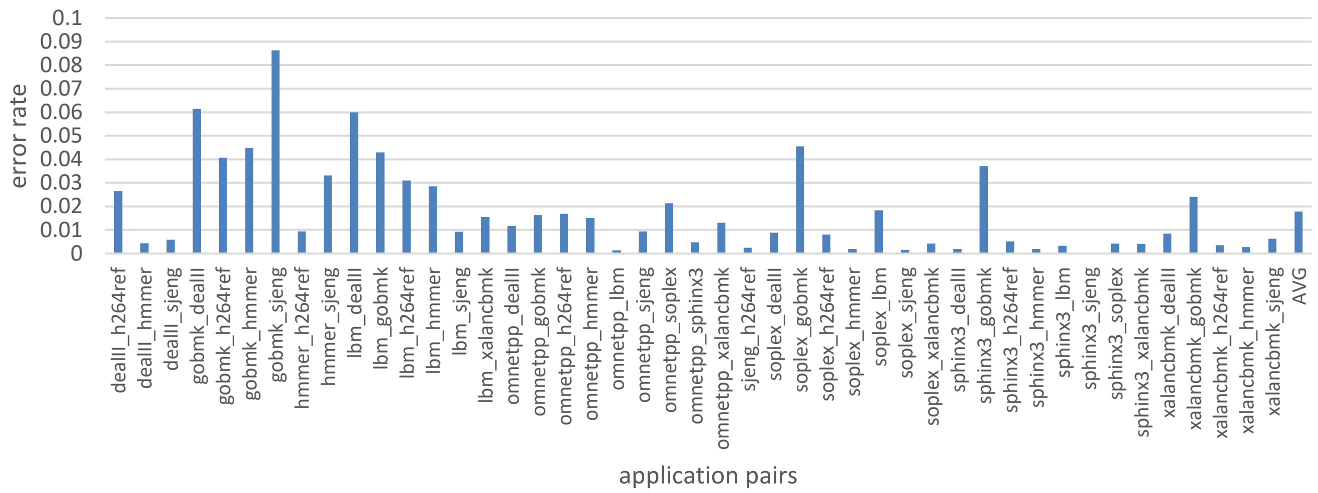


Fig. 16. The error rates of the miss count estimations of 45 application pairs for one-level shared cache architecture.

Table 3 lists the cache configurations with various sizes and way-associativity used in the test. In Table 3, the two numbers $x_1\_x_2$ on the first row indicate that the L_1 (private instruction and data) cache is of size x_1 bytes and the shared L_2 cache is of size x_2 bytes. Next, the two numbers $y_1\_y_2$ on the second row indicate that the L_1 cache is y_1-way set associative and the shared L_2 cache is y_2-way set associative. The symbol F indicates a fully associative cache.

Fig. 17 shows the error rates of the shared L_2 miss counts estimated by our approach as compared with the golden Sniper simulation results. In general, the geometric mean error rate of 45 application pairs is less than 2.4%.

The extremely low error rates support the fact that our proposed aggregated reuse-distance-base computation model is very accurate for multi-core and multi-level-cache designs. Therefore, once the cache size of each level is determined at the early design stage, we can effortlessly compute the cache miss rate of each cache hierarchical level based on the aggregated reuse-distance histograms. Henceforth, we may optimize cache configuration for best system performance for any given target applications. The experiment is discussed below.

4.4 Optimal Multi-core Two-Level Cache Designs

To show that our proposed method can effectively determine optimal cache designs, we compare our results with that of the partitioning approach discussed in section 3.3. Assume that the objective function is simply to minimize the total miss count of the shared last-level cache, L_2.

For comparison, we implement the partitioning method on a design with 8-KB private L_1 cache and 1-MB L_2 shared-cache. We find that by varying L_1 and L_2 cache sizes, we always obtain better performance results than that of the partitioning method regarding total execution cycle counts. The partitioning method focuses on minimizing the miss counts of L_2, or equivalently the number of main memory accesses, to increase performance. In fact, the miss count of L_1 also affects the performance, and our approach considers the total effect and hence get the best results. The access latencies of the L_1, L_2 caches, and main memory are the same as that used in the last section, i.e., one cycle, ten cycles and 130 cycles respectively.

Specifically, for the application pair *dealII* and *sjeng*, by increasing the L_1 cache to 16 KB while decreasing the L2

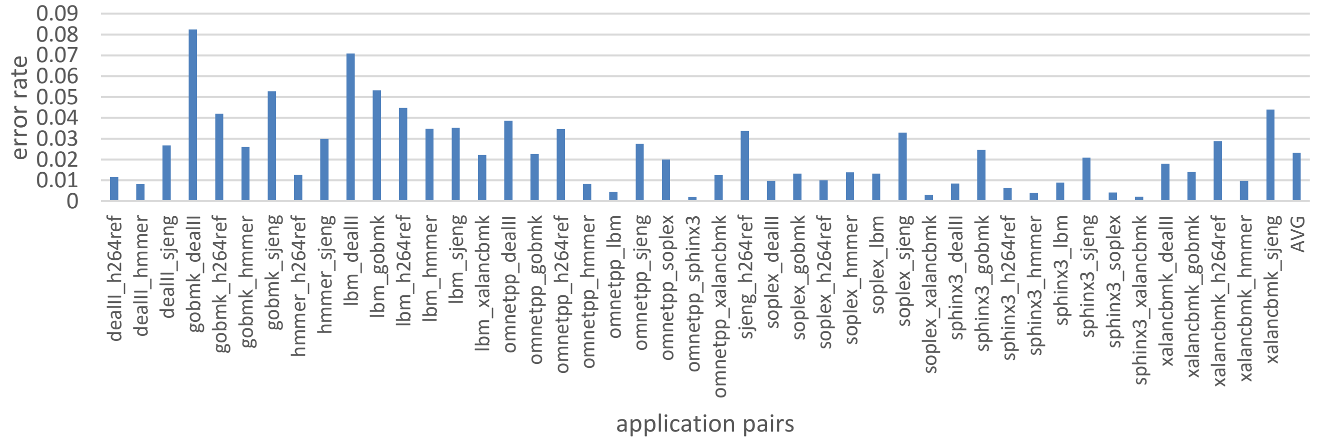


Fig. 17. The miss count estimation errors of 45 application pairs for a two-level shared cache architecture.

cache to 512 KB, the total number of cycle count to complete the execution is 7.24% less than the best partitioning results on the originally given cache sizes.

In general, the partitioning method requires hardware modification and has additional hardware cost. With the accurate aggregated reuse-distance computation model, we can now decide the optimal cache size of multi-level cache at the early design phase without requiring an unnecessary cost. After verification of our approach through simulations, we will next verify using real chips.

4.5 Test on Commercial SoCs

To check if the approach work for real chips, we choose a commercial Rockchip RK3288 SoC [34] for testing. The RK3288 has a quad-core ARM Cortex A17 [31] with 1M bytes L2 shared cache and an ARM Mali GPU [31] with 256K bytes L2 shared-cache. Both the L2 caches are last level caches, and 4-way set associative with a 16-byte cache line. We compute the L2 cache hit ratios of ARM Cortex A17 and Mali GPU from hardware cache performance counters.

The application used for testing is the Deep Learning Neural Networks (DNN) [32] for classifying images based on the darknet framework [33]. The application has two versions, one is implemented to run on CPU and another to run on GPU. The CPU case is a 4-thread DNN program, and the GPU case is a 256-threads DNN program specifically optimized for GPU. Each thread computes independent data.

In table 4, the first column lists four DNN architectures, Darknet reference, Darknet 19, Resnet 50 and Densenet 201 used for testing. The aggregate reuse-distance is computed based on each CPU or GPU core computes independent data. There is average 3% L2 hit ratio difference between the real hardware device and our approach. For the GPU testing case, the difference of L2 hit ratio is 9%, so using reuse distance to estimate the shared-cache hit ratios for different cache designs is effective.

Table 4: Comparing the estimated and real measured L2 Cache hit ratios.

	Our Estimation		Real Measure	
	GPU	CPU	GPU	CPU
Darknet reference	73.27%	84.61%	65.36%	82.95%
Darknet19	71.47%	82.81%	63.57%	79.12%
Resnet50	69.75%	83.52%	64.52%	80.50%
Densenet 201	69.48%	83.8%	64.01%	80.85%

4.6 Discussions

The error sources of our approach mainly are contributed by the following three factors. First, the major one is due to the estimation error of the proposed aggregated reuse-distance histogram generation method. Nevertheless, the experiments show that the average error rate is less than 2%. As discussed in the main body of the paper, we may conduct simulations of the target application pairs to get more accurate histogram data, but then we will need to assume certain cache sizes and run long simulations to obtain the results. However, fixing cache sizes will then defeat the purpose of being able to determine the optimal cache sizes at the early system-level design phase. Therefore, the estimation-induced error is a choice of our tradeoff decision.

Secondly, the invalidated LRU records on the higher-level inclusive cache also contributed to errors. Note that for an inclusive cache architecture, the data on an L_1 cache always has a replica in L_2. When a victim data of L_1 is evicted and replaced, the replica is still in L_2 with an outdated LRU value, which theoretically should be inherited from the corresponding L_1 victim data, following most practiced cache implementations. Nevertheless, the inconsistency contributes slight inaccuracy.

Finally, since real designs are mostly of k-way associativity, not the ideal fully associativity, the modeling discrepancy also causes errors. However, as verified in Section 4.1 and 4.6 experiments, the results show that the above two factors contribute only negligible errors.

5 Conclusion

In this paper, we have proposed an accurate aggregated reuse-distance computation model to estimate the shared-cache hit/miss counts without needs to run extensive simulations. Using a simple scanning search approach, we efficiently and effectively explore multi-level cache configurations for optimization. Our proposed approach does not require hardware support or OS modification and is perfect for early system design use. For future work, we may extend the idea to perform complete storage system optimization and test heterogeneous systems including CPU-GPU or CPU-DSP.

References

[1] Basu, Arkaprava, et al. "Scavenger: A new last level cache architecture with global block priority." Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2007. M. K.

[2] Qureshi, Moinuddin K., et al. "Adaptive insertion policies for high performance caching." ACM SIGARCH Computer Architecture News. Vol. 35. No. 2. ACM, 2007.

[3] Jaleel, Aamer, et al. "High performance cache replacement using re-reference interval prediction (RRIP)." ACM SIGARCH Computer Architecture News. Vol. 38. No. 3. ACM, 2010.

[4] Khan, Samira, Yingying Tian, and Daniel Jiménez. "Sampling dead block prediction for last-level caches." Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on. IEEE, 2010.

[5] Duong, Nam, et al. "Improving cache management policies using dynamic reuse-distances." Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2012.

[6] Qureshi, Moinuddin K., and Yale N. Patt. "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches." Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2006.

[7] Xie, Yuejian, and Gabriel H. Loh. "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches." ACM SIGARCH Computer Architecture News. Vol. 37. No. 3. ACM, 2009.

[8] Kim, Seongbeom, Dhruba Chandra, and Yan Solihin. "Fair cache sharing and partitioning in a chip multiprocessor architecture." Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society, 2004.

[9] Chandra, Dhruba, et al. "Predicting inter-thread cache contention on a chip multi-processor architecture." High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on. IEEE, 2005.
[10] Xu, Chi, et al. "Cache contention and application performance prediction for multi-core systems." Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010.
[11] Sandberg, Andreas, David Black-Schaffer, and Erik Hagersten. "Efficient techniques for predicting cache sharing and throughput." Proceedings of the 21st international conference on Parallel architectures and compilation techniques. ACM, 2012.
[12] Liu, Chun, Anand Sivasubramaniam, and Mahmut Kandemir. "Organizing the last line of defense before hitting the memory wall for CMPs." Software, IEE Proceedings-. IEEE, 2004.
[13] Brock, Jacob, et al. "Optimal cache partition-sharing." Parallel Processing (ICPP), 2015 44th International Conference on. IEEE, 2015.
[14] Chang, Jichuan, and Gurindar S. Sohi. "Cooperative cache partitioning for chip multiprocessors." ACM International Conference on Supercomputing 25th Anniversary Volume. ACM, 2014.
[15] Suh, G. Edward, Larry Rudolph, and Srinivas Devadas. "Dynamic partitioning of shared cache memory." The Journal of Supercomputing 28.1 (2004): 7-26.
[16] Mattson, Richard L., et al. "Evaluation techniques for storage hierarchies." IBM Systems Journal 9.2 (1970): 78-117.
[17] Subramanian, Lavanya, et al. "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory." Proceedings of the 48th International Symposium on Microarchitecture. ACM, 2015.
[18] Eklov, David, David Black-Schaffer, and Erik Hagersten. "Fast modeling of shared caches in multicore systems." Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers. ACM, 2011.
[19] Chen, Xi E., and Tor M. Aamodt. "Modeling cache contention and throughput of multiprogrammed manycore processors." Computers, IEEE Transactions on61.7 (2012): 913-927.
[20] Chen, Xi E., and Tor M. Aamodt. "A first-order fine-grained multithreaded throughput model." High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on. IEEE, 2009.
[21] Carlson, Trevor E., Wim Heirman, and Lieven Eeckhout. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation." Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011.
[22] Henning, John L. "SPEC CPU2006 benchmark descriptions." ACM SIGARCH Computer Architecture News 34.4 (2006): 1-17.
[23] Jaleel, Aamer. "Memory characterization of workloads using instrumentation-driven simulation." Web Copy: http://www. glue. umd. edu/ajaleel/workload(2010).
[24] Cheng-Lin Tsai, et al. "A Fast-and-Effective Early-Stage Multi-level Cache Optimization Method Based on Reuse-Distance Analysis." National Tsing Hua University, 2016.
[25] Jaleel, Aamer, et al. "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches." High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on. IEEE, 2015.
[26] Wilton, Steven JE, and Norman P. Jouppi. "CACTI: An enhanced cache access and cycle time model." Solid-State Circuits, IEEE Journal of 31.5 (1996): 677-688.
[27] http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html
[28] Niu, Qingpeng, et al. "PARDA: A fast parallel reuse distance analysis algorithm." Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE, 2012.
[29] Hamerly, Greg, et al. "Simpoint 3.0: Faster and more flexible program phase analysis." Journal of Instruction Level Parallelism 7.4 (2005): 1-28.
[30] Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." ACM Sigplan Notices. Vol. 40. No. 6. ACM, 2005.
[31] http://www.arm.com
[32] Schmidhuber, J. et al. "Deep Learning in Neural Networks: An Overview." Neural Networks. 2015.
[33] https://github.com/pjreddie/darknet
[34] http://www.rock-chips.com/

Hsin-I Wu received the B.S. and M.S. degree from the Department of Electrical Engineering, National Taiwan University, Taiwan, in 1997 and 1999. He is currently pursuing the Ph.D. degree in computer science at National Tsing-Hua University, Taiwan. His current research interests include full system simulation and embedded system verification.

Chi-Kang Chen received the B.S. degree from Chung Hua University, Taiwan, in 1999, the M.S. degree from National Dong Hwa University, Taiwan, in 2002, all in computer science and information engineering. He is currently pursuing a Ph.D. degree in computer science at National Tsing Hua University, Taiwan. Chi-Kang's academic interests include computer architecture, embedded system, system level design, and memory system. He is also a senior engineer in the Information and Communications Research Laboratories (ICL) of Industrial Technology Research Institute (ITRI), Taiwan.

Hsin-Yu Ho received her B.S. degree in computer science from National Chung Hsing University, Taiwan in 2012. And she received her M.S. degree from National Tsing Hua University, Taiwan in 2017. She worked on network communication development in embedded systems.

Chia-Chi Lee received his B.S. degree in computer science from National Chung Hsing University, Taiwan in June 2017. He is currently an M.S. student at the National Tsing Hua University, Taiwan. He is a member of Logos Lab and focuses research on memory architectures for Deep Neural Networks.

Ren-Song Tsay Ren-Song Tsay (M'06) received the Ph.D. degree from UC Berkeley, Berkeley, CA, USA. He was with the IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, before he started his successful Silicon Valley ventures. He designed the first commercially successful performance optimization physical design system (now in Synopsys) which is still the market leader. He then jointly founded Axis Systems (now merged with Cadence) and developed a breakthrough logic verification system using reconfigurable computer technology. After that, he helped a few start-up companies as a consultant or investor. He is currently teaching at National Tsing-Hua University, Taiwan, on the subjects of high-tech entrepreneurship and system level design.