LWPTool: A Lightweight Profiler to Guide Data Layout Optimization

Article *in* IEEE Transactions on Parallel and Distributed Systems · May 2018

DOI: 10.1109/TPDS.2018.2840992

CITATIONS
3

READS
57

5 authors, including:

Chao Yu
Beihang University (BUAA)
8 PUBLICATIONS **21** CITATIONS
SEE PROFILE

Yuebin Bai
Beihang University (BUAA)
20 PUBLICATIONS **43** CITATIONS
SEE PROFILE

LWPTool: A Lightweight Profiler to Guide Data Layout Optimization

Chao Yu, Probir Roy, Yuebin Bai, *Member, IEEE*, Hailong Yang, *Member, IEEE*, and Xu Liu

Abstract—Memory access latency continues to be a dominant bottleneck in a large class of applications on modern architectures. To optimize memory performance, it is important to utilize the locality in the memory hierarchy. Data layout optimization can significantly improve memory locality. However, pinpointing inefficient code and providing insightful guidance for data layout optimization is challenging. Existing tools typically leverage heavyweight memory instrumentations, which hinders the applicability of these tools for real long-running programs. To address this issue, we develop LWPTool, a profiler to pinpoint top candidates that benefit from data layout optimization. LWPTool makes three unique contributions. First, it adopts lightweight address sampling to collect and analyze memory traces. Second, LWPTool employs a set of novel methods to determine memory access patterns to guide data layout optimization. We also formally prove that our method has high accuracy even with sparse memory access samples. Third, LWPTool scales on multithreaded machines. LWPTool works on fully optimized, unmodified binary executables independently from their compiler and language, incurring around 6.2 percent runtime overhead. To evaluate LWPTool, we study ten sequential and parallel benchmarks. With the guidance of LWPTool, we are able to significantly improve all these benchmarks; the speedup is up to 1.39× on average.

Index Terms—Data locality, address sampling, lightweight profiling, structure splitting, array regrouping

1 Introduction

Modern processors employ a hierarchy of caches to reduce the data access latency to main memory. Usually, caches physically located nearer to the CPU have higher data transfer bandwidth and lower access latency. To efficiently utilize the memory hierarchy, one needs to maintain good data locality to avoid frequent accesses to a far away memory. Memory access patterns that block data into the fastest caches and access it multiple times before it is evicted are said to have excellent data locality. There are two typical types of data locality: temporal and spatial. An access pattern exhibits temporal locality when it accesses the same memory words multiple times. An access pattern exploits spatial locality when it accesses a memory location and then accesses nearby locations soon afterward. Typically, spatial locality goes unexploited when accessing data with a large stride or indirection. While hardware prefetching can recognize non-unit accesses, long access strides cause low utilization of caches due to frequent capacity-based evictions.

Access patterns with poor spatial locality cause useless data to be loaded into cache. Such access patterns squander much of a processor's memory bandwidth loading values from main memory that will never be used before being evicted. Fig. 1 shows an example. In Fig. 1a, the original code creates an array of structure Arr but uses only two fields in one loop and the other two fields in a second loop. As all the four structure fields, a, b, c, and d per array element reside in the same cache line, the computation in both loops only uses part of a cache line, wasting significant cache space and memory bandwidth. In Fig. 1b, each iteration loads as many as n cache lines into cache as it reads an element from each of n distinct arrays. If n is large, the cache line accesses during each iteration conflict with each other and cause conflict misses. To address above performance issues, one can split structures [1] to optimize data layouts in memory—this approach is referred to as "structure splitting". In addition, multiple arrays can be regrouped [2] into one structure array to improve spatial locality and reduce cache contention—this approach is referred to as "array regrouping". With above two optimizations, only useful data is loaded into the cache in each loop, which efficiently uses cache space and bandwidth.

Identifying opportunities for data layout optimization and making appropriate decisions for code transformation are difficult, especially for the programs that consist of hundreds of thousands of code lines and run with hundreds of threads. Developers need insightful guidance from compilers or performance tools to make decisions. There are principally two techniques to identify data affinity: static analysis and dynamic profiling. Static analysis [2], [3], [4], [5], which generates data layout optimization decisions at compilation time, has three weaknesses. First, it cannot accurately handle pointers and aliases that are widely used for memory references. Second, it is difficult to provide optimization advice for the whole program which has multiple source files compiled separately. Third, it relies on specific compilers instead of working on the fully optimized binary code.

- *C. Yu, Y. Bai, and H. Yang are with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China. E-mail: {yuchao, byb, hailong.yang}@buaa.edu.cn.*
- *P. Roy and X. Liu are with the Department of Computer Science, College of William and Mary, Williamsburg, VA 23185. E-mail: {proy, xl10}@cs.wm.edu.*

Manuscript received 19 Sept. 2017; revised 11 Feb. 2018; accepted 22 May 2018. Date of publication 28 May 2018; date of current version 10 Oct. 2018. (Corresponding author: Hailong Yang.)
Recommended for acceptance by F. Cappello.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2840992

```
struct type {int a; int b; int c; int d;};
struct type Arr[N];
for (i = 0; i < N ; i++)
    B[i] = Arr[i].a + Arr[i].c;
        ...
for (i = 0; i < N; i++)
        C[i] = Arr[i].b + Arr[i].d;

//splits the structure Arr into two: Arr1 and Arr2.
struct type1 {int a; int c;};
struct type2 {int b; int d;};
struct type1 Arr1[N];
struct type2 Arr2[N];
for (i = 0; i < N; i++)
    B[i] = Arr1[i].a + Arr1[i].c;
        ...
for (i = 0; i < N; i++)
        C[i] = Arr2[i].b + Arr2[i].d;
```

(a) The optimized code splits the structure Arr into two for better spatial locality in both loops.

```
int A1[N], A2[N], ...,An[N];
for (i = 0; i < N; i++)
    D[i] = A1[i] + A2[i] + ... + An[i];

//regroup arrays A1, A2, ..., An into Arr
struct type3 {int A1; int A2; ...; int An;};
struct type3 Arr[N];
for (i = 0; i < N; i++)
    D[i] = Arr[i].A1 + Arr[i].A2 + ... + Arr[i].An;
```

(b) Arrays A1, A2, ..., An are regrouped into Arr. Arr[i].An now represents An[i].

Fig. 1. The code and data structures before and after structure splitting and array regrouping optimization.

By contrast, dynamic profiling technique overcomes these shortcomings. However, many existing methods [1], [6], [7] incur high overhead. To reduce the overhead, researchers have explored bursty sampling [8] to monitor a small subset of memory accesses as a representative of the whole program. Bursty sampling, however, still incurs 3-5× overhead [9], which hinders its applicability for long-running applications.

To address these limitations, we develop LWPTool, a lightweight profiler that guides data layout optimization. LWPTool makes the following contributions:

- LWPTool adopts an efficient sampling-based data collection mechanism based on hardware performance monitoring units, which reduces the runtime overhead of LWPTool to ∼6.2 percent on average.
- LWPTool employs a set of novel methods to provide insightful guidance for data layout optimization. Such methods include filtering out insignificant data, accurately identifying field accesses, computing latency-based field affinities and evaluating regrouping benefits.
- LWPTool works for both sequential and parallel applications, with scalable designs.

LWPTool works on fully optimized binary executables without manual instrumentation. LWPTool is independent from compilers and programming languages. To evaluate LWPTool, we study ten well-known benchmarks, covering both sequential and multi-threaded code. Guided by LWPTool, we are able to identify structure fields or arrays with high affinities in the whole program. LWPTool provides insightful guidance to optimize these benchmarks with structure splitting and array regrouping, and yields a 1.39× speedup on average.

TABLE 1
The Address Sampling Techniques in Different Processor Models and their Successors

Processor models	Sampling techniques
Intel Nehalem	Precise event-based sampling with load latency (PEBS-LL)
Intel Itanium	Data event address register (DEAR)
Intel Pentium4	Precise event-based sampling (PEBS)
AMD Opteron	Instruction-based sampling (IBS)
IBM POWER5	Marked event sampling (MRK)

We organize the remaining paper as follows. Section 2 introduces the background knowledge about the technique LWPTool uses for efficient memory profiling. Section 3 reviews the existing work on data layout optimization. Section 4 shows the overview of LWPTool. Section 5 describes the details of LWPTool that employs novel analysis techniques in providing data layout optimization guidance. Section 6 shows the design and implementation of LWPTool that employs novel analysis techniques. Section 7 provides the evaluation results of LWPTool with ten representative benchmarks. Section 8 concludes this work.

2 Background

To support lightweight analysis, LWPTool leverages address sampling that is available in most modern CPU architectures. In address sampling, performance monitoring units (PMUs) periodically select a memory access and monitor its execution through the pipeline. Table 1 lists the address sampling mechanisms in modern architectures. Typically, address sampling captures three pieces of information: (1) the instruction pointer (IP) of the sampled memory access, (2) the memory address (effective address) the sampled instruction reads or writes, and (3) related memory events caused by the sampled instruction, such as cache or TLB misses. Among these sampling techniques, only PEBS-LL and IBS provide latency measure for the sampled memory access, which is necessary for LWPTool. Thus, LWPTool is built atop both Intel PEBS-LL and AMD IBS.

Advantages of Address Sampling. Compared to traditional measurement techniques, address sampling provides unique capabilities. Unlike instrumentation-based measurement [10],[9], address sampling incurs extremely low overhead, less than 3 percent. Because of the negligible distortion to the monitored program execution, the observed memory events (e.g., cache misses and latency) related to the sampled access can precisely quantify the execution bottlenecks. By contrast, to assess the cache misses and their latency, instrumentation-based methods require simulators, but are difficult to model complex memory architectures. Unlike event-based sampling (EBS) supported by traditional performance counters, address sampling records more information. EBS does not record the instruction pointer that triggers the event or the effective address. There is also no latency information captured. Thus, address sampling outperforms existing measurement techniques and provides potential opportunities for efficient and effective analysis of memory bottlenecks.

Challenges with Address Sampling. There are two challenges for the analysis based on the data collected by address sampling. First, address sampling only monitors a small subset

of memory accesses. Thus, it requires some novel methods to extract access patterns hidden in the raw data to represent the profile of the whole program execution. Second, address sampling randomly monitors memory accesses, which has less flexibility than the instrumentation technique. With instrumentation, one can leverage the bursty sampling technique [8] to monitor all memory accesses in a code region or a time window, which facilitates access pattern analysis [9], [11]. However, address sampling does not support monitoring bursty memory accesses. Instead, the distance (in the number of memory accesses) between the two adjacent memory samples varies and is usually large. Thus, LWPTool cannot directly apply prior methods for access pattern analysis. In Section 5, we introduce a set of novel methods to show their effectiveness and accuracy in access pattern analysis to guide data layout optimization.

3 Related Work

There are numerous prior efforts focusing on data layout optimization in memory, but we only discuss the ones that are closely related to LWPTool. Section 3.1 reviews the prior approaches on structure splitting optimization and array regrouping optimization; Section 3.2 reviews the lightweight memory profiling based on address sampling.

3.1 Data Layout Optimization

There are a number of compiler-based techniques [1], [2], [3], [4], [5], [12], [13], [14], [15] to perform data layout optimization. They either rely on static analysis or extra information generated by compilers. There are three weaknesses with compiler-based techniques compared to the dynamic analysis. First, static analysis has limited capability in handling aliases and pointers. Second, compilers usually analyze the code at a granularity of files so it is difficult for them to give optimization advice based on the whole program. Third, the compiler-dependent features impede the usage. Therefore, we only concentrate on the existing work of dynamic analysis in following sections.

3.1.1 Structure Splitting Optimization

To be independent from compilers, Chilimbi et al. [6] used field access frequency to compute field affinities. Zhong et al. [1] guide structure splitting by quantifying the affinity between structure fields via collecting reuse distance signatures of each structure field. Similar to LWPTool, their approach performs whole-program profiling that guarantees that the optimization can benefit the whole program execution rather than individual loops. However, computing the reuse distance incurs overhead [16] as high as 153×.

Yan et al. [7] developed ASLOP to identify structure splitting opportunities with the help of code instrumentation and hardware performance counters. It counts the execution frequency of basic blocks and cache misses to compute structure field affinities. ASLOP reduces overhead by saving instrumentation on every memory access. However, its overhead is still as high as 4.2×.

In contrast to these approaches, LWPTool does not instrument memory accesses or basic blocks, so it has much lower runtime overhead. Yan et al. [17] developed the most related work to LWPTool, which leverages the performance monitoring units to monitor the basic block execution with low overhead. Unlike LWPTool, their approach does not directly analyze memory addresses to quantify the affinities between fields of a data structure. Thus, their approach requires substantial manual efforts to identify structure splitting candidates. Moreover, their approach does not work for parallel programs.

Wang et al. [18] proposed on-the-fly structure splitting for heap data objects. They maintain a customized heap and leverage online data copying to perform structure splitting. However, their approach splits all the fields in a structure rather than computing their affinities, as known as maximal splitting [19], [20], which may lead to sub-optimal performance. Moreover, they do not perform whole program analysis to guide the structure splitting.

3.1.2 Array Regrouping Optimization

Seidl and Zorn [21] proposed a profile-driven approach to identify and segregate objects based on lifetime and reference density so that the most frequently referenced data can be regrouped to improve virtual memory performance. Zhong et al. [1] defined a profiling-based method to measure how close a group of data is accessed together. Their profiling method generates better regrouping decisions than static analysis. However, the latest work [16] from their group shows that their footprint profiling method leads to 113× slowdown on average. Besides, the frequency-based metric is not accurate. Because the higher frequency does not mean greater importance.

Shen et al. [22] presented an inter-procedural analysis technique for array regrouping. They used two methods to analyze array affinities: a compiler-based method and a lightweight profiling-based method. Jiang et al. [23] extended their approach to work for multithreading applications. However, their lightweight profiling-based method still causes about 100 percent overhead for all programs they tested.

Marin et al. [11] proposed an approach based on static analysis and simulation to analyze memory access patterns of array streams. They grouped arrays, whose access patterns are amenable to hardware prefetching, together to increase the prefetching effectiveness. However, their approach may regroup any arrays meet the condition, which does not analyze the feasibility of array regrouping. Moreover, the simulation method incurs more than 60× slowdown compared to native execution.

The aforementioned approaches for identifying regrouping opportunities have some weaknesses. All of these approaches do not pinpoint significant arrays and take all arrays of a program into consideration, which increases analysis overhead. In contrast, LWPTool filters out insignificant arrays, because optimizing these arrays yields little performance gain. In addition, the frequency-based affinity metric, known as the state-of-the-art approach, is misleading, because the higher frequency does not always mean greater importance. LWPTool uses the latency-based metric to compute affinity, which gives a more accurate guidance.

3.2 Lightweight Memory Profiling

Buck and Hollingsworth developed Cache Scope [24] to perform data-centric analysis using Itanium 2 EAR. Cache Scope associates latency with data objects and functions

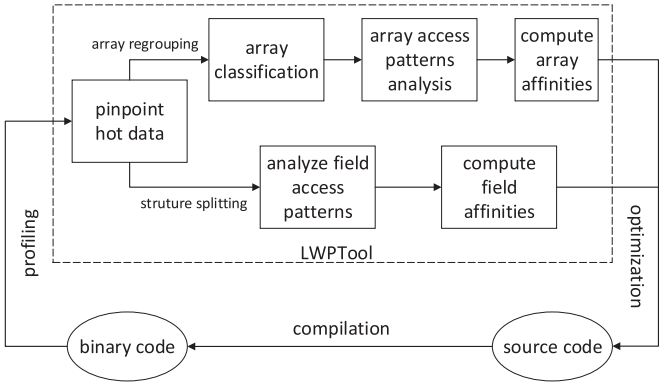


Fig. 2. The workflow of LWPTool's analysis. LWPTool collects performance profiles during execution and analyzes these profiles to provide performance insights.

that access them. HPCToolkit [25], [26] enhances Cache Scope by attributing latency metrics to the full calling contexts of code and data. HPCToolkit collects data based on AMD IBS and IBM MRK. Memphis [27] and MemProf [28] leverages AMD IBS to associate latency with data structures to identify costly memory accesses to remote sockets.

Unlike LWPTool, these tools only collect and attribute address samples, they do not perform sophisticated, automatic pattern recognition of the sampled memory accesses. Some recent tools such as HPCToolkit-NUMA [29], ArrayTool [30], and ScaAnalyzer [31] perform analysis to guide data placement across sockets, array regrouping, and memory scaling losses, respectively. However, none of these tools can give advice for both structure splitting and array regrouping. To the best of our knowledge, *LWPTool is the first lightweight profiling tool that provides insightful guidance for both structure splitting and array regrouping in both sequential and parallel programs.*

In addition to data layout optimization, similar memory profiling technique is also used for data placement optimization [32], [33] in hybrid/heterogeneous memory (HM) systems. Although data layout optimization and data placement optimization have the same final goal of improving memory performance, they are very different approaches. Data layout optimization mainly explores better data locality for hot memory regions, whereas data placement optimization aims to reduce memory access latency by allocating the costly memory regions with higher bandwidth storage. In general, these two approaches are complementary to each other.

4 LWPTool Methodology

In this section, we describe our approach which guides data layout optimization with the data collected by address sampling. Fig. 2 shows an overview of the analysis workflow of LWPTool. First, we filter out insignificant data from consideration to avoid wasting optimization efforts on inconsequential parts of the program. Then, for structure splitting, we analyze field access patterns to identify how different fields of a data structure are referenced in each loop; after that we aggregate access pattern analysis from all loops in the program and compute field affinities for the whole program. For array regrouping, we classify arrays into different groups according to their lifetimes, locations and schemes to avoid the useless analysis on infeasible arrays. Next, we analyze array access patterns to identify arrays with access pattern conflict. Finally, we compute array affinities for the whole program. Programmers are required to do two tasks to achieve end-to-end optimization: (1) compile the source code with any compiler and any optimization option[1] and (2) transform the source code with the data layout optimization guidance.

Before diving into the details of each analysis component, we first introduce the preprocessing stages with address samples, which are the foundation of other analysis. We preprocess address samples by associating them with code-centric and data-centric views during the program execution.

Code-Centric Attribution. Associating samples with code, such as instructions and loops, is important for LWPTool's analysis. It is straightforward to attribute samples to instructions, as the performance monitoring unit (PMU) captures the instruction pointer for each sample. To identify loops surrounding each memory access, LWPTool parses the machine code for a program and then uses interval analysis [34] to identify loop boundaries. With information about loops, we can attribute samples whose instruction pointers fall in the loop interval.

Data-Centric Attribution. LWPTool leverages existing techniques [26] to associate address samples with data objects. It records memory ranges allocated for data objects by reading symbols and overloading allocation functions. The names of static data objects in the symbol table and the allocation call paths for heap data objects are used to uniquely identify data objects. LWPTool does not monitor stack data objects, which usually cause trivial performance impact in the memory. Then, LWPTool leverages the effective address captured by PMU to attribute the sample to a data object. For data-centric attribution, LWPTool does not distinguish array of aggregate types (structures) or array of primitive types.

Along with the sample attributions, all events, such as cache misses and latency, associated with samples are aggregated to a specific line of code or a data object. Based on these two sample attributions, LWPTool performs the analysis in Fig. 2. We describe each analysis in the following sections.

5 Data Layout Optimization Guidance

5.1 Pinpointing Hot Data

LWPTool filters out data objects that have low access latency during the whole program execution, because optimizing these data yields little performance gain. LWPTool first monitors the memory allocation to identify data objects used in the program. LWPTool only monitors memory allocated on the static and heap section currently. Data objects in stack can be changed into static or heap objects for examination.

For data objects allocated in the static data section of a load module, such as executable program and shared library, LWPTool parses the symbol table of the module to get the memory range of each static data object. To record the life cycle of each static data object, LWPTool also tracks the loading and unloading of each load module. When a module is loaded, its static data objects are allocated; and all static data objects are freed when the module is unloaded.

1. -g is needed to map the analysis to the source code.

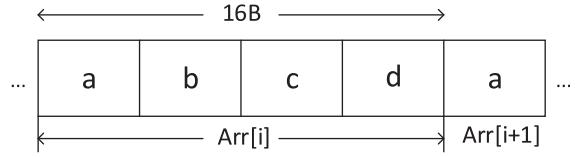


Fig. 3. The stride of accessing a data structure field in an array of structure. This is an example for the first loop in Fig. 1a Accessing field a with $Arr[i].a$ has a stride of the whole structure size, 16 bytes.

To monitor data objects allocated dynamically in the heap by one of the *malloc* family of functions (*malloc*, *calloc*, *realloc*) during the execution of the module, LWPTool overloads the allocation and free functions. When the allocation functions are called in the module, LWPTool enters into the overloaded functions and extracts the pointer to the allocated memory and the memory size. Besides, the full calling context is obtained using call stack unwinding [26] to help locate where the data is allocated in the program, which also uniquely identifies the heap data object.

After identifying the data objects used in the program, LWPTool then monitors data accesses using the address sampling technology introduced in Section 2. With data-centric attribution, the memory access latency of each sampled memory access is associated with a data object. Thus, LWPTool computes a metric l_d, as shown in Equation (1), for data object.

$$l_d = \frac{\sum_{data} l}{\sum_{program} l} \tag{1}$$

In Equation (1), l is the latency of a sampled access. $\sum_{data} l$ and $\sum_{program} l$ sum up latency associated with the investigated data object and the whole program respectively. l_d is between 0 and 1, where a higher value means more significance of the data object. This data-centric profiling helps LWPTool to focus its analysis only on significant data objects.

5.2 Structure Splitting

5.2.1 Analyzing Field Access Patterns

LWPTool performs access pattern analysis for each significant data structure in each individual loop. The goal of this analysis is to identify which fields of the examined data structure are accessed in each loop. As discussed in Section 2, without monitoring every memory access, it is challenging to perform this analysis on binary with sparse samples: there is no easy way to know the structure size and there is no easy way to know which field of this structure is accessed by a specific memory instruction. LWPTool obtains this information by analyzing address samples based on one key observation:

> Observation: *If a memory instruction keeps accessing one field of an aggregate data structure across loop iterations, this instruction shows a non-unit, constant stride access pattern. This stride is (a multiple of) the size of the structure.*

Understanding this observation is straightforward. Taking the first loop in Fig. 1a as an example, the memory access of $Arr[i].a$ has a stride of 16 bytes because the access needs to cross $Arr[i].b$, $Arr[i].c$, and $Arr[i].d$ to touch $Arr[i+1].a$ as shown in Fig. 3. In addition, if loop induction variable i increments more than one per iteration, e.g., n, the stride for accessing field a is $16n$. From this observation, we can transform the analysis for the field access to stride analysis. If we can compute the stride, we can assess the data structure size. Moreover, from the offset computation (shown later), we can know which field is accessed.

To perform the analysis, LWPTool makes the following assumption: an instruction in a specific calling context only accesses one field of a data structure. For a formal description, if an instruction i accesses an array starting at address A and this array object is of the structure type defined as fields $f1$, $f2$, ..., fn, instruction i in one calling context, with this assumption, only accesses field fi of this array of structure. In practice, this assumption holds most of the time on a large variety of code, especially scientific solvers. Based on this assumption, we identify streams and analyze their stride (Section 5.2.2).

Identifying Streams. We define a *stream* as a sequence of memory accesses to an array in a loop, corresponding to the loop induction variable. Based on the assumption, we simply say that any instruction in a loop that accesses a data structure is a stream. If the same instruction accesses multiple data structures, this instruction forms multiple streams. With the help of code- and data-centric attributions, we can identify all the streams in a loop and associate the instruction pointers as well as the involved data structures with them. A stream is a basic unit which we use for further analysis. Compared to the stream measurement by detailed memory instrumentation [11], our approach may incur some inaccuracy, but from our experiments, it can practically identify streams accurately with low overhead.

5.2.2 Analyzing Strides

Stride analysis computes the structure size and identifies the field that is accessed by each stream based on address samples. We develop a novel method—the GCD algorithm—in LWPTool for efficient stride analysis. We formally describe the GCD algorithm as follows:

Suppose m_1, m_2, ...m_k are k samples with unique addresses of a stream that references data object D in loop L. These k samples access different offsets of D. We use Equation (2) to compute the address difference d between each two adjacent sampled addresses.

$$d_i = |m_i - m_{i-1}|(where, 1 < i \leq k) \tag{2}$$

Then we compute the greatest common divisor (GCD) of these address differences as the stride of accessing D in loop L using Equation (3).

$$stride = gcd(d_i)(where, 1 < i \leq k) \tag{3}$$

Algorithm Intuition. We take the array of structure Arr in Fig. 3 for example. If a stream always accesses field a (e.g., $Arr[i].a$ in a loop with induction variable i), the two adjacent samples can access $Arr[m].a$ and $Arr[n].a$; the difference between the two sampled addresseses is obviously a multiple of the structure size, 16 bytes. With taking more samples (e.g., sampled $Arr[2].a$, $Arr[5].a$, and $Arr[7].a$), we expect the GCD of the address differences of these samples (e.g., 48 and 32 bytes) to be 16 bytes, reflecting the real stride of the access pattern.

Accuracy Analysis. To formally quantify the accuracy of the GCD algorithm, we use $stride_r$ to denote real stride of the stream that accesses D in loop L. According to the definition of stride, $stride$ computed in Equation (3) is always a multiple of $stride_r$. For example, if only the addresses of $Arr[2].a$, $Arr[4].a$, and $Arr[6].a$ are sampled, $stride$ is computed as 32 rather than the $stride_r$ 16. To have $stride$ equal to $stride_r$ with high probability, there should be enough address samples for this stream. We then prove that, with a small number of samples per stream, the GCD algorithm can obtain $stride_r$ with a very high probability. To not lose generality, we quantify the accuracy for analyzing a loop with unit ($stride_r$ = 1) stride. Equation (4) shows the accuracy of our stride analysis method with k samples of $unique$ addresses out of total n addresses. $\binom{\frac{n}{i}}{k}$ is the number of ways all k samples leading to the computation of stride as i, instead of 1.

$$\begin{aligned} accuracy &= 1 - \frac{\binom{\frac{n}{2}}{k} + \binom{\frac{n}{3}}{k} + \binom{\frac{n}{5}}{k} + \cdots}{\binom{n}{k}} \\ &< 1 - \frac{1}{2^k} - \frac{1}{3^k} - \frac{1}{5^k} - \cdots \end{aligned} \tag{4}$$

The equation subtracts all the possibilities of computing $stride$ that is not equal to 1. We only consider i of the prime multiples of stride 1, because it also covers all the cases for composite multiples. From this equation, we can see that *if k is larger than 10, the accuracy can be higher than 99 percent*. For real stride $stride_r$ of different values, we can get a similar equation and conclusion. 10 unique address samples is a significantly small number for data structures that are frequently accessed. Hence, our approach ensures high accuracy.

It is worth noting that the GCD algorithm reports irregular access patterns with stride 1, not distinguishing from the unit stride. However, access patterns with stride 1, either regular or irregular, are not of interest for LWPTool because there is no structure splitting opportunity.

Computing Structure Size. As the type and size of a data structure does not change during the program execution, we aggregate all the streams that access the same data structure and calculate the structure size by taking GCD of their strides, as shown in Equation (5). In this equation, $stride_i$ is the stride computed for stream i in the whole program that is associated with this data structure. The accuracy analysis is similar to the aforementioned GCD algorithm.

$$size = gcd(stride_i). \tag{5}$$

Identifying Accessed Structure Field. We use the offset of the field to the starting address of the data structure to identify it uniquely. For example, the offsets of fields a and b in the structure $type$ in Fig. 1a are 0 and 4 bytes, respectively. We compute this offset for each stream with Equation (6). In this equation, m_i is the effective address of an arbitrary memory access in stream i; s is the starting address of the data structure in the memory, which is obtained from either monitoring data allocation functions or reading from symbol tables; $size$ is the size of the data structure obtained from Equation (5).

$$offset_i = (m_i - s) \bmod size. \tag{6}$$

5.2.3 Computing Field Affinities

We compute the field affinities of data structures for the whole program execution. We compute the affinities between all fields in each significant data structure. Unlike previous approaches that count the number of memory accesses, we use the memory access latency to derive an affinity metric, which gives a more accurate memory access penalty to each structure field. We use two steps to compute the affinity metric.

First, we analyze each loop. We aggregate the latency incurred by all the streams in a loop that have the same offset in the same data structure, for the purpose of quantifying the accesses to a structure field in a loop. Second, we take all loops in the program into consideration and use Equation (7) to compute the affinity FA_{ij} between field i and j of a structure.

$$FA_{ij} = \frac{\sum flc_{ij}}{\sum fl_{ij}}. \tag{7}$$

In Equation (7), $\sum flc_{ij}$ is the aggregate latency of accessing field i and j in all the common loops that reference both fields, while $\sum fl_{ij}$ is the total latency of accessing fields i and j in the whole program. Intuitively, when fields i and j are always accessed together in loops, $\sum flc_{ij}$ is a large proportion of $\sum fl_{ij}$. F_{ij} is between 0 and 1; the larger FA_{ij} is, the higher affinity is between stream i and j. We compute the affinity between each field pair. We cluster all fields with high affinities in a structure and generate the structure splitting guidance: regroup the structure fields with high affinities.

5.3 Array Regrouping

5.3.1 Array Classification

To find the regrouping opportunity is different from to find the splitting opportunity described in previous section. Different arrays may have different lifetimes, locations and schemes, so, regrouping arbitrary arrays is not always feasible. To find arrays can be regrouped together, LWPTool classifies arrays in the program according to lifetime, location and scheme. Only arrays of the same lifetime, location and scheme can be classified together and further may have the opportunity to be regrouped together.

Lifetime. The lifetime of an array is the duration from the array is allocated to freed. If the lifetimes of two arrays overlap, then the arrays have the same lifetime. However, there is no need to require two arrays to be absolutely equal in lifetime. Instead, LWPTool use the *active time* of each array. The active time of an array begins when the array is first sampled and ends when the array is last sampled. If the active time of each array is covered by the lifetime of another, then LWPTool considers the arrays for regrouping.

Location. Only arrays in the same module can be regrouped, arrays allocated in different modules cannot be regrouped. In addition, arrays of different types also cannot be regrouped, such as a static array and a heap-allocated array.

Scheme. Two arrays have the same scheme if they have the same number of elements, otherwise cannot be regrouped. To get the scheme of each array, LWPTool first uses the method of computing structure size described in Section 5.2.2 to calculate the element size of each array. Then the element number of each array can be computed with Equation (8).

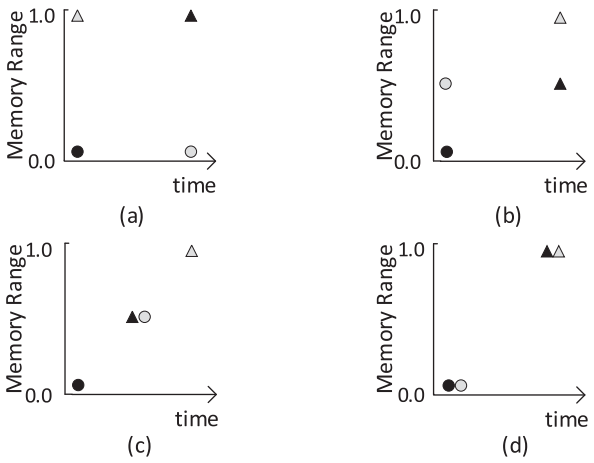


Fig. 4. Four array accesses patterns. The gray and black shapes represent two arrays accessed together in a loop. Circular and triangle are the lower and upper bounds of relative access offsets. (a) two arrays are accessed in reverse order; (b) different parts of the two arrays are accessed together; (c) different parts of the two arrays are accessed at different time; (d) two arrays are accessed in the same order.

$$scheme_i = \frac{memsize_i}{size_i}. \tag{8}$$

In Equation (8), $scheme_i$ is the element number of array i, which is also the scheme of array i; $memsize_i$ is the memory size of array i, which is obtained with the methods described in Section 5.1; $size_i$ is the element size of array i. LWPTool does not distinguish one-dimensional array or multi-dimensional array, because the access pattern of a multi-dimensional array can be easily changed to one-dimensional mode in most cases.

After array classification, significant arrays are classified into different groups. LWPTool then does further analysis of each array group.

5.3.2 Analyzing Array Access Patterns

LWPTool filters out arrays that have conflict access patterns in each group. There is no benefit to regroup arrays that have conflict access patterns even if they exhibit high affinities with each other. To identify conflicts, LWPTool analyzes the access pattern of each significant array in all loops. For the analysis, LWPTool uses code-centric attribution to get the information of which loop each sample belongs to; besides, which array each sample accesses should also be obtained using the data-centric attribution of LWPTool. For each sampled memory access m to array i in the loop l, LWPTool computes its relative access offset o in i using Equation (9).

$$o = \frac{m_a - i_{begin}}{i_{end} - i_{begin}}. \tag{9}$$

In Equation (9), m_a is the memory address accessed by m; i_{begin} and i_{end} are the boundaries of the memory range allocated for i. Because m_a is one address of array i, m_a is always between [i_{begin}, i_{end}] and thus o is always between [0, 1]. LWPTool clusters o of all the sampled accesses to array i in loop l by computing a [i_{min}, i_{max}] interval of o, where i_{min} and i_{max} are the lower and upper bounds of o in loop l. In addition to relative access offset o, LWPTool also records the access timestamp of each sample. Then LWPTool determines whether two arrays conflict or not according to their lower and upper bounds of o and corresponding timestamps. As shown in Fig. 4, if the lower and upper bounds of o of two arrays are different or in reverse order, the two arrays conflict in access pattern. LWPTool uses Equation (10) to judge whether two arrays conflict or not.

$$C_{ij} = \begin{cases} true & i_{max} \le j_{min} \ or \ j_{max} \le i_{min}; \\ true & t^i_{min} \ge t^j_{max} \ or \ t^j_{min} \ge t^i_{max}; \\ false & other. \end{cases} \tag{10}$$

In Equation (10), i and j are two arrays in the same group; i_{min} and i_{max} are the lower and upper bounds of o of array i in loop l, and t^i_{min} and t^i_{max} are the corresponding timestamps; j_{min} and j_{max} are the lower and upper bounds of o of array j in loop l, and the corresponding timestamps are t^j_{min} and t^j_{max}; and C_{ij} is the conflict state between i and j. If array i is in conflict with j, then the value of C_{ij} is $true$; otherwise C_{ij} is $false$.

It should be noticed that the address sampling technique used in LWPTool samples memory access at a fixed interval of events, thus LWPTool may not accurately report the boundaries of each array accessed in loops. However, from our experiments, such inaccuracy does not affect the results of the array access pattern analysis.

5.3.3 Computing Array Affinities

After above array classification and access pattern analysis, we compute the array affinities of each array group. LWPTool computes the affinities between all arrays that are not conflicting in each array group. We use a similar two-step method as Section 5.2.3 to compute the array affinity.

First, LWPTool aggregates the latency incurred by all samples in a loop that access to the same array to quantify the accesses to this array. Second, LWPTool takes all loops into consideration and use Equation (11) to compute the affinity AA_{ij} between arrays i and j.

$$AA_{ij} = \frac{\sum alc_{ij}}{\sum al_{ij}}. \tag{11}$$

In Equation (11), $\sum alc_{ij}$ is the aggregate latency of accessing arrays i and j in all loops that access both arrays; $\sum al_{ij}$ is the total latency of accessing arrays i and j in the whole program. AA_{ij} is between 0 and 1. Obviously, if arrays i and j are always accessed together in loops, AA_{ij} will be or close to 1.

Consequently, LWPTool suggests two arrays i and j with high AA_{ij} and without access pattern conflict are final regrouping candidates.

5.4 Handling Parallel Programs

To adapt the analysis for parallel programs with multiple threads or/and processes, we perform the aforementioned analysis for individual thread/process online and then merge the analysis results offline. LWPTool adopts a technique similar to HPCToolkit [26] to merge the code- and data- centric attributions in different threads/processes. For code-centric attribution, we aggregate the metrics (e.g., latency) for the sampled memory accesses from different profiles with the same instruction pointer. For data-centric attribution, we aggregate the metrics with memory of the

TABLE 2
Benchmark Descriptions

Benchmarks	Application Descriptions	Parallel
179.ART [38]	Neural Network based object recognision in a thermal image	No
462.libquantum [39]	Simulation of quantum computer	No
TSP [40]	Traveling Salesman Problem solver	No
Mser [41]	Image analyser for face detection	No
CLOMP 1.2 [42]	Designed to measure OpenMP and multi-threading performance issues	Yes
Health [43]	Columbian health care simulation	Yes
NN [44]	Find k-nearest neighbour from unstructured data set	Yes
SRAD [44]	A diffusion method for ultrasonic and radar imaging applications	Yes
lavaMD [44]	Calculates particle potential and relocation between particles	Yes
IRSmk [45]	Solves a diffusion equation on a 3D, block structured mesh	Yes

same allocation site or the same name. A user can view the aggregate execution profile in a code- or data-centric manner, focusing either on hot code regions or hot memory regions. Based on the sample attributions, we perform all the analysis in Fig. 2. For identifying significant memory regions, we use the aggregate latency from different profiles to compute l_d for each memory. For analyzing access patterns, we adapt Equation (5) to use access strides computed in different profiles to calculate the structure size and array element size. For computing field and array affinities, all latency metrics are computed with the aggregate latency from different profiles.

6 LWPTool Implementation

LWPTool consists of an online profiler and an offline analyzer. The profiler accepts a binary executable and monitors its execution. The analyzer processes the raw data collected during the profiling and associates the analysis with program structures and source code to provide guidance for data layout optimization. The remaining sections elaborate on the implementation of LWPTool's profiler and analyzer, and discuss the challenges we address for parallel programs.

6.1 Online Profiler

LWPTool uses libmonitor [35] to preload the profiling library into the address space of the binary executable. The libmonitor tool provides function callbacks before and after the program execution. In the program begin callback, the profiler sets up address sampling, while in the program end callback, the profiler reclaims all the resources allocated for the sampling and writes the profile into a file for further analysis. When a predefined number of marked events occur, an address sample is triggered and the profiler uses an interrupt handler to capture it. The handler records the instruction pointer, effective address, and the data latency associated with each sample. Moreover, to avoid the high overhead incurred by monitoring every memory allocation, LWPTool can be configured to monitor memory allocations that are larger than a predefined threshold. Therefore, the profiler has very low runtime overhead. The overhead of LWPTool is evaluated in Section 7.

With these raw data, the profiler performs online code- and data-centric attribution. With the help of hpcstruct, a tool from HPCToolkit, the profiler identifies loops and attributes samples. To attribute samples to memory regions, the profiler uses libmonitor to overload memory allocation functions and uses symtabAPI [36] to read symbol tables from the binary. The profiler performs the GCD algorithm online to compute the stride for each stream.

Scalability. To scale the data collection and online analysis of the profiler, we design the profiler to monitor each thread individually to avoid any thread synchronization operation during data collection and attribution, which minimizes the runtime overhead and achieves scalable measurement. Thus, LWPTool efficiently attributes samples in parallel programs to code and memory regions. LWPTool analyzes access patterns for each thread separately and writes the analysis result to a profile file per thread.

6.2 Offline Analyzer

LWPTool's analyzer accepts the profiles generated by the profiler. It computes field and array affinities based on the access patterns collected by the profiler. The analyzer extracts the binary-to-source code line mapping information from debugging sections recorded in the binary and maps the analysis results, such as the allocation of memory regions and significant loops, to the source code. Finally, LWPTool's analyzer outputs this high-level analysis as advice for data optimization.

LWPTool's offline analyzer needs to merge all of the profiles from different threads and processes. If the number of threads and processes is huge, merging their profiles can be time-consuming. To expedite this process, LWPTool leverages the reduction tree algorithm [37] to merge all profiles in parallel.

LWPTool can be easily configured to do either the structure splitting analysis or array regrouping analysis. In general, if there are structures frequently used in loops, then structure splitting analysis is preferred for performance optimization, otherwise if there are arrays frequently used in loops, then the array regrouping analysis is more useful to find the performance potential. It would be effective if the user is familiar with programs to be analyzed by LWPTool. However, if the user has little knowledge about the program, both structure splitting and array regrouping can be applied to identify the performance opportunity with both techniques.

7 Experimental Evaluation

We evaluate LWPTool on an Intel Xeon machine that has 16 2.10 GHz Xeon E5-2620 v4 cores. The machine has a private 32 KB L1 data cache and a private 256 KB L2 cache for each core, as well as a 20 MB shared L3 cache. We study ten sequential and parallel benchmarks from a few well-known suites, as shown in Table 2. All these benchmarks are compiled with gcc 4.8 -g -O3. LWPTool uses Intel PEBS-LL PMUs to take an address sample for every 10,000 memory accesses (analysis of different sampling intervals is discussed in Section 7.11). We run all parallel benchmarks with four threads in the same socket to avoid performance impact from multiple memory controllers and socket interconnects.

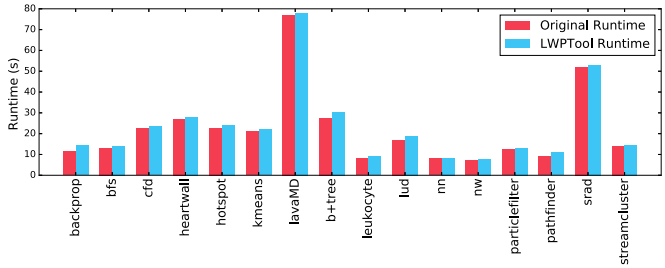


Fig. 5. The runtime overhead of LWPTool when monitoring Rodinia benchmarks.

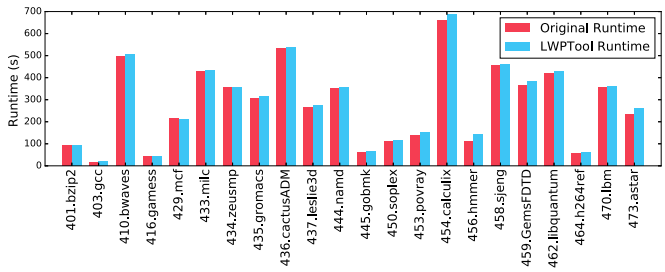


Fig. 6. The runtime overhead of LWPTool when monitoring SPEC CPU 2006 benchmarks.

TABLE 3
Execution Speedups after Data Layout Optimization Guided by LWPTool and the Measurement Overhead Incurred by LWPTool

Benchmarks	Opt. type	Original exec. time	Optimized exec. time	Speedups	LWPTool overhead
179.ART	Splitting	14.2s	9.9s	1.43×	3.2%
462.libquantum	Splitting	419.2s	200.6s	2.09×	2.1%
TSP	Splitting	42.9s	38.3s	1.12×	21.4%
Mser	Splitting	31.2s	29.6	1.05×	3.7%
CLOMP 1.2	Splitting	22.6s	19.4s	1.16×	8.8%
Health	Splitting	45.6s	40.2s	1.13×	12.3%
NN	Splitting	8.0s	3.5s	2.29×	5.0%
SRAD	Regrouping	52.3s	37.2s	1.41×	1.3%
lavaMD	Regrouping	77.1s	68.5s	1.13×	1.6%
IRSmk	Regrouping	88.7s	80.1s	1.11×	2.7%
Average	–	–	–	**1.39×**	**6.2%**

For each data, we report its average value from three executions.

LWPTool Overhead. We use LWPTool to monitor all Rodinia [44] and SPEC CPU 2006 [39] benchmarks and report the runtime overhead in Figs. 5 and 6. For Rodinia, we use the default datasets released with the suite; for SPEC CPU 2006, we use the reference inputs. Our experimental results show that LWPTool incurs only ∼8.6 and ∼4.7 percent overhead on average for Rodinia and SPEC CPU 2006 benchmarks respectively.

Optimization Overview. Data layout optimization is more effective for memory intensive applications. Table 3 provides an overview of the speedups for each benchmark from the data layout optimization guided by LWPTool. All benchmarks receive significant improvements in their end-to-end execution time, as high as 1.39× on average. For libquantum and NN, most of their execution time is spent on memory accesses, therefore the speedup (more than 2×) is much higher than the rest of the benchmarks. For TSP, LWPTool incurs over 20 percent runtime overhead due to the large number of memory allocation operations. In general, LWPTool incurs only ∼6.2 percent runtime overhead on average to collect the performance data for the optimization guidance of the reported benchmark applications.

TABLE 4
Cache Miss Reduction after Optimizing Benchmarks with LWPTool

Benchmarks	L1 miss reduction	L2 miss reduction	L3 miss reduction	TLB miss reduction
179.ART	44.0%	36.1%	32.1%	36.5%
462.libquantum	50.7%	61.5%	97.2%	1.5%
TSP	29.6%	20.7%	31.2%	52.3%
Mser	14.2%	19.3%	41.1%	22.7%
CLOMP 1.2	25.1%	24.9%	24.6%	52.6%
Health	73.6%	89.1%	1.2%	7.3%
NN	87.3%	87.5%	50.0%	66.0%
SRAD	26.1%	23.0%	-12.5%	58.1%
lavaMD	-30.2%	6.3%	3.4%	35.3%
IRSmk	85.0%	4.2%	28.8%	99.8%

TABLE 5
LWPTool's Access Pattern Analysis of ART to Identify Fields of $f1\_neuron$ Structure

Field	I	W	X	V	U	P	Q	R
Latency	4.2%	12.3%	3.1%	12.3%	5.1%	58.7%	4.1%	0.2%

LWPTool also associates the access latency to each field in the percentage of the total latency accessing this structure.

To verify that most of the performance gains are from the improvement of data locality, we collect the hardware performance counter measurements for all benchmarks. Table 4 shows that the most data cache misses are substantially reduced in each data cache level and TLB. Although the L3 misses for SRAD and the L1 misses for lavaMD are increased, their data TLB misses are reduced by a large margin, which reduce the opportunity of the time-consuming page table walk. Overall, the results show how structure splitting and array regrouping improve program performance: they enhance data locality.

In the following sections, we show the case studies of LWPTool on each benchmark. Note that although only one analysis mechanism is applied for each benchmark in the case studies, the two mechanisms can be used together if there are performance opportunities for both structure splitting and array regrouping. For instance, after structure splitting the array regrouping can be used to find significant arrays (including original arrays and arrays generated by structure splitting) to be regrouped.

7.1 SPEC CPU 2000 - ART

With the help of the data-centric attribution and the derived latency metric l_d, LWPTool identifies that an array of structure, $f1\_neuron$, used in ART accounts for 86.2 percent of total memory access latency. LWPTool further decomposes the latency to different fields of $f1\_neuron$ when analyzing the access patterns, as shown in Table 5. Each field, except R, accounts for significant latency; among them, field P is the hottest, with 58.7 percent of total latency.

We show LWPTool's code-centric analysis in Table 6. It associates latency with each significant loops and identifies the fields accessed in each loop. From this table, we can see which fields are always accessed together in hot loops. For example, the hottest loop of line 615-616 accounts for 51.6 percent of total

TABLE 6
LWPTool Associates Latency with Structure Fields of $f1\_neuron$ in each Monitored Loop of ART

Loops with line number	Latency percentage	Accessed fields
132-138	0.7%	U,P
545-548	8.2%	I,W,U
553-554	10.7%	W
559-570	6.5%	X,Q
575-576	11.3%	V
584-600	0.5%	U,P
607-608	10.3%	P
615-616	51.6%	P
1015-1016	0.2%	I

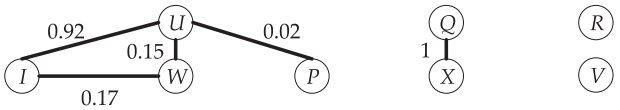


Fig. 7. LWPTool generates the affinity information for the fields in structure $f1\_neuron$ of ART.

```
typedef struct {double P;} f1_neuron_P;
typedef struct {double Q; double X;} f1_neuron_XQ;
typedef struct {double U; double *I;} f1_neuron_IU;
typedef struct {double V;} f1_neuron_V;
typedef struct {double W;} f1_neuron_W;
typedef struct {double R;} f1_neuron_R;
```

Fig. 8. Structure splitting of $f1\_neuron$ of ART.

latency and only field P is accessed in this loop. Another hot loop in line 559-570, both field X and Q are accessed.

LWPTool quantifies field affinities in Fig. 7. We see that fields I and U have a high affinity, 0.92, so LWPTool recommends to group the two fields in a new structure. Although fields I and W have an affinity 0.17, and fields U and W have an affinity 0.15, the loop (as shown in Table 6) accessing field W alone is weighted than the loop accessing the three fields together. Thus, fields W is not recommended to group with fields I and U. The fields X and Q also show a high affinity, which also suggests to group them in a new structure. On the other hand, although P and U are accessed together in two loops, they have a low affinity 0.02, because these two loops account for a smaller amount of latency compared to the loops that only access P.

According to the analysis of LWPTool, we split the structure of $f1\_neuron$ to six new structures, as shown in Fig. 8. This optimization yields a 1.43× speedup.

7.2 SPEC CPU 2006 - Libquantum

For libquantum, LWPTool identifies an array of the type $quantum\_reg\_node\_struct$ accounts for 99.8 percent of total memory access latency. $quantum\_reg\_node\_struct$ has two fields in this structure: $amplitude$ and $state$; accessing field $state$ accounts for ∼100 percent of the access latency of this structure. LWPTool's code-centric analysis pinpoints three hot loops at line 89-98, 170-174 and 61-66 that access $state$ and respectively account for 41.8, 39.0 and 19.2 percent of the total latency incurred by $quantum\_reg\_node\_struct$. As these hot loops do not access $amplitude$, LWPTool reports the affinity between these two fields as 0. According to LWPTool's analysis, we split structure $quantum\_reg\_node\_struct$ as shown in Fig. 9, which leads to a 2.09× speedup.

```
struct quantum_reg_node_struct_a {
    COMPLEX_FLOAT amplitude;};
struct quantum_reg_node_struct_s {
    MAX_UNSIGNED state;};
struct quantum_reg_node_struct {
    struct quantum_reg_node_struct_a * struct_a_ptr;
    struct quantum_reg_node_struct_s * struct_s_ptr;};
```

Fig. 9. Structure splitting of $quantum\_reg\_node\_struct$ in libquantum.

```
struct tree_0 { double x,y; int next;};
struct tree_1 { int sz; int left, right; int prev;};
```

Fig. 10. The optimized structure of $tree$ after structure splitting in TSP Benchmark.

```
idx_t *GNode_parent_pt;
typedef struct {idx_t shortcut; idx_t region;
  int area;} node_t;
```

Fig. 11. Structure splitting for $node\_t$ in MSER.

7.3 Olden - TSP

For TSP, we have optimized the process of tree building in order to store all nodes in several arrays rather than performing one memory allocation operation for each node. LWPTool identifies that arrays with the $tree$ structure account for 100 percent of the total memory access latency. The structure has seven fields: sz, x, y, $left$, $right$, $next$, $prev$. LWPTool further shows that fields x, y and $next$ account for 15.6, 9.3 and 75.1 percent of the entire access latency to this data structure. In addition, LWPTool pinpoints that these three fields are accessed together in the loops at line 139-142 and 170-173 in tsp.c, associated with 27.5 and 72.5 percent of the total access latency, respectively. LWPTool quantifies that these three fields have high affinities of 1 and gives recommendations to group them into one new structure. Thus, we split these fields from the rest of the four fields in tree as shown in Fig. 10 and achieve a speedup of 1.12× for the whole program.

7.4 SD-VBS - MSER

LWPTool identifies three arrays used in MSER but only the array of $node\_t$ is significant, accounting for 21.7 percent of total memory access latency incurred during the program execution. LWPTool's stride analysis pinpoints field $parent$ alone (offset 0 with stride 16) is frequently accessed in a hot loop at line 679-683. Thus, we split the structure to separate $parent$ as shown in Fig. 11 and obtain a 1.05× speedup of the whole program.

7.5 CORAL - CLOMP

CLOMP is designed to measure OpenMP overhead and performance impact due to parallel threads. We run CLOMP with four OpenMP threads. LWPTool's latency metric l_d shows that the array of the structure type $\_Zone$ incurs ∼100 percent of the total memory latency throughout the benchmark execution. This array is allocated by one thread but accessed by all of threads in parallel. LWPTool merges the accesses to this array from different threads. With further analysis, LWPTool shows that the loop at line 328-337 accounts for all access of this array. Stride analysis suggests that only fields $value$ and $nextZone$ are accessed in this loop. Field $value$ accounts for 49.8 percent of the total access latency of the loop, while $nextZone$ accounts for 50.2 percent. LWPTool computes the affinity between these two fields as 1

```
struct _ZoneHeader {long zoneId; long partId;};
struct _Zone {double value;
      struct _Zone *nextZone;
      struct _ZoneHeader *head;
};
```

Fig. 12. The optimized structure of *_Zone* after structure splitting in CLOMP Benchmark.

```
struct Patient_super_struct {
  struct Patient_super_struct *forward;
  struct Patient *this; };
struct Patient {
  int id; int32_t seed; int time; int time_left;
  int hosps_visited; struct Village *home_village;
  struct Patient_super_struct *back;};
```

Fig. 13. Structure splitting for *Patient* in health.

```
struct neighbor_entry { char entry[RECLENGTH];};
struct neighbor_dist { double dist;};
```

Fig. 14. Structure splitting for *neighbor* in NN.

TABLE 7
LWPTool Identifies Arrays used in SRAD

Group	Line number	Array	Latency percentage
1	72	J	15.3%
	73	c	18.0%
	81	dN	16.3%
	82	dS	13.1%
	83	dW	23.8%
	84	dE	12.4%
-	71	I	0.0%
-	75	iN	0.0%
-	76	iS	0.1%
-	77	jW	0.6%
-	78	jE	0.5%

Arrays are grouped using method described in Section 5.3.1.

but 0 affinity with the other two fields *zoneId* and *partId*. LWPTool recommends to split this structure and group *value* and *nextZone* together. Fig. 12 illustrates the data structure after splitting. We create a new field *head* to keep track of the two fields *zoneId* and *partId* that are split to a new structure *_ZoneHeader*. As a result we achieve a 1.16× speedup for the whole CLOMP Benchmark.

7.6 BOTS - Health

Health is implemented with OpenMP tasking constructs. We run Health with four threads. LWPTool identifies two arrays of structure but its derived latency metric shows that only *Patient* is significant, accounting for 97.1 percent of total memory access latency. Structure *Patient* has eight fields. Further access pattern analysis by LWPTool shows that only field *forward* is accessed in a hot loop at line 96. The affinity analysis reports that *forward* has low affinities with other fields so it recommends splitting field *forward* from other fields. Fig. 13 shows the resulting structure splitting. This suggested optimization leads to a speedup of 1.13×.

7.7 Rodinia - NN

NN is an OpenMP program. LWPTool monitors eight arrays but only one array of the structure *neighbor* accounts for 98.9 percent of total access latency. Structure *neighbor* has two fields: *entry* and *dist*. LWPTool reports that fields *dist* and *entry* respectively account for 99.7 percent and 0.3 percent of the total access latency to this structure. Further analysis by LWPTool reveals that the hot loop at line 117-120 references *dist* but not *entry*. The affinity between the two fields is 0, so LWPTool recommends to split the two fields for better performance. Fig. 14 shows the result of structure splitting. We optimize NN with the guidance of LWPTool and obtain a 2.29× speedup for the whole program.

TABLE 8
LWPTool Associates Latency with Arrays in each Monitored Loop of SRAD

Loops with line number	Latency percentage	Accessed arrays
172-187	88.4%	J, c, iS, jE, dN, dS, dW, dE
136-162	11.6%	J, c, iS, jW, jE, dN, dS, dW, dE

TABLE 9
LWPTool Computes Affinities between any two of the Detected Arrays in SRAD

Affinities	J	c	dN	dS	dW	dE
J	1	-	-	-	-	-
c	0.89	1	-	-	-	-
dN	1	1	1	-	-	-
dS	1	1	1	1	-	-
dW	1	1	1	1	1	-
dE	0.85	1	1	1	1	1

```
typedef struct {
      float J;  float c;  float dN;
      float dS; float dW; float dE;
} JcdNSWEGroup;
```

Fig. 15. Array regrouping for arrays J, c, dN, dS, dW and dE in SRAD.

7.8 Rodinia - SRAD

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations. It is parallelized with OpenMP. We show LWPTool's code-centric analysis in Tables 7 and 8. Table 7 shows the arrays identified by LWPTool and associates latency with each detected array. LWPTool classifies arrays J, c, dN, dS, dW and dE as a group. From the analysis of the source code of SRAD, although array I should be classified as the same group as arrays J, c, dN, dS, dW and dE, there is not enough information for LWPTool to do that. In general, these arrays cannot be classified are usually insignificant.

As shown in Table 8, only two loops account for ~100 percent total memory access latency of SRAD. From this table, we can see which arrays are always accessed together in hot loops. For example, arrays J, c, iS, jE, dN, dS, dW and dE are always accessed together in the two hot loops. Because of the low latency percentage, LWPTool does not do further analysis on arrays iS, jE and jW.

Array affinities quantified by LWPTool are shown in Table 9. We see that arrays J, c, dN, dS, dW and dE have high affinities (1, 0.89 and 0.85) between each other. Therefore, LWPTool suggests that all of these six arrays should be regrouped as shown in Fig. 15. The regrouping result shows

```
for (...) {
  for (k = 0; k < (1 + box[l].nn); k ++) {
    ......
    first_j = box[pointer].offset;
    ......
    for (i = 0; i < NUMBER_PAR_PER_BOX; i = i + 1) {
      for (j = 0; j < NUMBER_PAR_PER_BOX; j = j + 1) {
            r2 = rA[i].v + rB[j].v - DOT(rA[i],rB[j]);
                 ......
                 d.x = rA[i].x  - rB[j].x;
                 ......
                 fA[i].z +=  qB[j]*fzij;  }}}}
```

Fig. 16. The hottest loop nest in lavaMD. box points to box_cpu, rA and rB point to rv_cpu, qB points to qv_cpu and fA points to fv_cpu.

TABLE 10
Array Affinities in lavaMD Benchmark

Affinities	box_cpu	rv_cpu	qv_cpu	fv_cpu
box_cpu	1	-	-	-
rv_cpu	0	1	-	-
qv_cpu	0	1	1	-
fv_cpu	0	1	1	1

```
typedef struct {
  FOUR_VECTOR rv_cpu;
  fp qv_cpu;
  FOUR_VECTOR fv_cpu;} rqfv_cpu;
```

Fig. 17. Array regrouping for arrays rv_cpu, qv_cpu and fv_cpu in lavaMD.

1.47× speedup in running time for the whole SRAD benchmark.

7.9 Rodinia - lavaMD

lavaMD is an OpenMP benchmark, which calculates particle potential and relocation between particles in a three dimensional space. LWPTool identifies a loop nest (from line 117 to 188 in $kernel\_cpu.c$), shown in Fig. 16, and the innermost loop accounts for more than 99 percent of total memory access latency of lavaMD. LWPTool pinpoints four hot arrays box_cpu, rv_cpu, qv_cpu and fv_cpu, which are accessed together in the loop nest shown in Fig. 16. These four arrays respectively account for 20.4, 49.1, 9.6 and 20.9 percent of total memory access latency. Table 10 shows the array affinities calculated by LMPTool. We can see that affinities between them are all 1 while the affinities between box_cpu and other three arrays are 0. So, LWPTool recommends regrouping arrays rv_cpu, qv_cpu and fv_cpu. We optimize lavaMD with the guidance of LWPTool as shown in Fig. 17. In these regrouped arrays, arrays rv_cpu and fv_cpu are aggregate type and array qv_cpu is primitive type. The optimization obtains a 1.13× speedup for the whole program.

7.10 Sequoia - IRSmk

IRSmk solves a diffusion equation on a three-dimensional block-structured mesh. The loop kernel performs matrix multiplication in a three nested level. LWPTool identifies that this parallel loop has significant memory access latency. Further data-centric analysis shows that there are 27 arrays accessed together in the loop, which account for ∼100 percent of total memory access latency. LMPTool quantifies that these 27 arrays have high affinities of 1 with each other and gives recommendation to group them into one new structure. Thus, we group these arrays into one and achieve a speedup

```
typedef struct
{double dbl; double dbc; double dbr; double dcl;
 double dcc; double dcr; double dfl; double dfc;
 double dfr; double cbl; double cbc; double cbr;
 double ccl; double ccc; double ccr; double cfl;
 double cfc; double cfr; double ubl; double ubc;
 double ubr; double ucl; double ucc; double ucr;
 double ufl; double ufc; double ufr;} ABCDGroup;
```

Fig. 18. The regrouping result in IRSmk.

TABLE 11
Percentage of Relative Latencies for each Field of Structure $f1\_neuron$ and Total Latencies of Structure $f1\_neuron$ in ART Identified by LWPTool with Different Sampling Intervals

Intervals	Relative latency of each field								Total latency	Overhead
	I	W	X	V	U	P	Q	R		
2000	4.3%	9.4%	2.2%	8.5%	5.3%	67.6%	2.7%	0.1%	85.0%	24.5%
4000	4.8%	13.1%	2.9%	12.2%	5.4%	57.7%	3.7%	0.1%	85.6%	21.5%
6000	4.1%	9.7%	2.2%	8.7%	4.9%	67.8%	2.4%	0.1%	84.9%	14.1%
8000	4.3%	10.6%	2.4%	8.8%	5.1%	66.1%	2.5%	0.3%	86.4%	10.2%
10000	4.2%	12.3%	3.1%	12.3%	5.1%	58.7%	4.1%	0.2%	86.2%	3.2%
12000	4.4%	12.1%	2.8%	14.1%	5.5%	57.1%	4.0%	0.1%	86.7%	2.9%
14000	4.5%	9.5%	2.1%	8.1%	4.3%	68.6%	2.8%	0.1%	83.7%	2.6%

of 1.11× for the whole program. How the 27 arrays are regrouped is shown in Fig. 18.

7.11 Analysis of Sampling Intervals

In this section, we evaluate how different sampling intervals (in terms of memory accesses) affect LWPTool results. Table 11 shows the relative latencies of each field and total latencies of structure $f1\_neuron$ in benchmark ART identified by LWPTool with different sampling intervals from 2000 to 14000. It is clear that both relative latency of each field and the total latency of structure $f1\_neuron$ remain stable with different sampling intervals except the profiling overhead. The experiment results demonstrate that although decreasing the sampling interval increases the profiling overhead, it does not affect the accuracy of LWPTool to identify the problem code for performance optimization. Moreover, LWPTool can be easily configured by the user with customized sampling interval during the profiling stage. The experiments with other benchmarks show similar results and thus omitted here for brevity.

8 Conclusions

In this paper, we describe the design and implementation of a lightweight memory analysis tool, *LWPTool*, which identifies important data layout optimization opportunities—structure splitting and array regrouping—in both sequential and parallel programs. LWPTool adopts a set of novel methods to analyze address samples and solve challenges, such as accurate access pattern analysis based on sparse, random address samples. To evaluate LWPTool, we apply it on ten sequential and parallel benchmarks from well-known benchmark suites. The experimental results show that LWPTool incurs around 6.2 percent of runtime overhead, on average. Furthermore, LWPTool provides insightful guidance to optimize these benchmarks with structure splitting and array regrouping, and yields a 1.39× speedup on average. In our experience, it is easy to use LWPTool, and the feedback provided by LWPTool both visually and

quantitatively guide optimization of code performance. The manual effort in restructuring the code is minimal. LWPTool's output can be easily consumed by a compiler pass such as ROSE [46] to perform profile-guided data-layout optimization.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback and suggestion. This work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000503 and the National Science Foundation of China under Grant No. 61572062 and No. 61502019. This work is also supported by the National Science Foundation (NSF) under Grant No. 1464157. Hailong Yang is the corresponding author. Chao Yu and Probir Roy contribute equally to this paper.

References

[1] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, "Array regrouping and structure splitting using whole-program reference affinity," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 255–266, 2004.

[2] C. Ding and K. Kennedy, "Inter-array data regrouping," in *Proc. 12th Int. Workshop Languages Compilers Parallel Comput.*, 1999, pp. 149–163.

[3] V. D. L. Luz and M. Kandemir, "Array regrouping and its use in compiling data-intensive embedded applications," *IEEE Trans. Comput.*, vol. 53, no. 1, pp. 1–19, Jan. 2004.

[4] G. Chakrabarti, F. Chow, and L. PathScale, "Structure layout optimizations in the open64 compiler: Design, implementation and measurements," in *Proc. Open64 Workshop at Int. Symp. Code Generation Optimization*, 2008.

[5] N. Prashantha, T. Vikram, and N. Vaivaswatha, "Implementing data layout optimizations in the LLVM Framework," 2014, http://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," *ACM SIGPLAN Notices*, vol. 34, no. 5, 1999, pp. 13–24.

[7] J. Yan, J. He, W. Chen, P.-C. Yew, and W. Zheng, "ASLOP: A field-access affinity-based structure data layout optimizer," *Sci. China Inf. Sci.*, vol. 54, no. 9, pp. 1769–1783, 2011.

[8] Y. Zhong and W. Chang, "Sampling-based program locality approximation," in *Proc. 7th Int. Symp. Memory Manage.*, 2008, pp. 91–100.

[9] A. Rane and J. Browne, "Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics," in *Proc. 21st Int. Conf. Parallel Architectures Compilation Techn.*, 2012, pp. 147–156.

[10] K. Beyls and E. H. D'Hollander, "Refactoring for data locality," *Comput.*, vol. 42, no. 2, pp. 62–71, 2009.

[11] G. Marin, C. McCurdy, and J. S. Vetter, "Diagnosis and optimization of application prefetching performance," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, 2013, pp. 303–312.

[12] M. Hagog and C. Tice, "Cache aware data layout reorganization optimization in gcc," in *Proc. GCC Developers Summit*, 2005, pp. 69–92.

[13] R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in *Proc. Int. Symp. Code Generation Optimization*, 2006, pp. 233–244.

[14] J. Lin and P.-C. Yew, "A compiler framework for general memory layout optimizations targeting structures," in *Proc. Workshop Interaction between Compilers Comput. Archit.*, 2010, Art. no. 5.

[15] E. Raman, R. Hundt, and S. Mannarswamy, "Structure layout optimization for multithreaded programs," in *Proc. Int. Symp. Code Generation Optimization*, 2007, pp. 271–282.

[16] X. Xiang, C. Ding, H. Luo, and B. Bao, "HOTL: A higher order theory of locality," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 1, 2013, pp. 343–356.

[17] J. Yan, W. Chen, and W. Zheng, "PMU guided structure data-layout optimization," *Tsinghua Sci. Technol.*, vol. 16, no. 2, pp. 145–150, 2011.

[18] Z. Wang, C. Wu, P.-C. Yew, J. Li, and D. Xu, "On-the-fly structure splitting for heap objects," *ACM Trans. Archit. Code Optimization*, vol. 8, no. 4, 2012, Art. no. 26.

[19] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault, "MPADS: Memory-pooling-assisted data splitting," in *Proc. 7th Int. Symp. Memory Manage.*, 2008, pp. 101–110.

[20] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral, "Forma: A framework for safe automatic array reshaping," *ACM Trans. Program. Languages Syst.*, vol. 30, no. 1, 2007, Art. no. 2.

[21] M. L. Seidl and B. G. Zorn, "Segregating heap objects by reference behavior and lifetime," *ACM SIGPLAN Notices*, vol. 33, no. 11, 1998, pp. 12–23.

[22] X. Shen, Y. Gao, C. Ding, and R. Archambault, "Lightweight reference affinity analysis," in *Proc. 19th Annu. Int. Conf. Supercomputing*, 2005, pp. 131–140.

[23] Y. Jiang, E. Z. Zhang, X. Shen, Y. Gao, and R. Archambault, "Array regrouping on CMP with non-uniform cache sharing," in *Proc. 23rd Int. Conf. Languages Compilers Parallel Comput.*, 2010, pp. 92–105.

[24] B. R. Buck and J. K. Hollingsworth, "Data centric cache measurement on the Intel ltanium 2 processor," in *Proc. ACM/IEEE Conf. Supercomputing*, 2004, Art. no. 58.

[25] X. Liu and J. Mellor-Crummey, "Pinpointing data locality problems using data-centric analysis," in *Proc. 9th Annu. IEEE/ACM Int. Symp. Code Generation Optimization*, 2011, pp. 171–180.

[26] X. Liu and J. Mellor-Crummey, "A data-centric profiler for parallel programs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–12.

[27] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 87–96.

[28] R. Lachaize, B. Lepers, and V. Quéma, "MemProf: A memory profiler for NUMA multicore systems," in *Proc. ATC-USENIX Annu. Tech. Conf.*, 2012, pp. 5–5.

[29] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures," *ACM Sigplan Notices*, vol. 49, no. 8, pp. 259–272, 2014.

[30] X. Liu, K. Sharma, and J. Mellor-Crummey, "ArrayTool: A lightweight profiler to guide array regrouping," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn.*, 2014, pp. 405–415.

[31] X. Liu and B. Wu, "ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, Art. no. 47.

[32] D. Shen, X. Liu, and F. X. Lin, "Characterizing emerging heterogeneous memory," in *Proc. ACM SIGPLAN Int. Symp. Memory Manag.*, 2016, pp. 13–23.

[33] H. Servat, A. J. Pena, G. Llort, E. Mercadal, H.-C. Hoppe, and J. Labarta, "Automating the application data placement in hybrid memory systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 126–136.

[34] P. Havlak, "Nesting of reducible and irreducible loops," *ACM Trans. Program. Languages Syst.*, vol. 19, no. 4, pp. 557–567, 1997.

[35] M. W. Krentel, "Libmonitor: A tool for first-party monitoring," *Parallel Comput.*, vol. 39, no. 3, pp. 114–119, 2013.

[36] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, 2000.

[37] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw. Storage Anal.*, 2010, pp. 1–11.

[38] SPEC CPU2000 Benchmark Suite. (2005). [Online]. Available: http://www.spec.org/cpu2000

[39] SPEC CPU2006 Benchmark Suite. (2007). [Online]. Available: http://www.spec.org/cpu2006

[40] M. C. Carlisle, "Olden: Parallelizing programs with dynamic data structures on distributed-memory machines," Ph.D. dissertation, Princeton University, Princeton, NJ, USA, 1996.

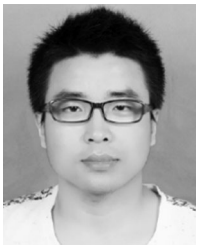
[41] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The san diego vision benchmark suite," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 55–64.

[42] LLNL Coral Benchmarks. (2014). [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks

[43] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Proc. Int. Conf. Parallel Process.*, 2009, pp. 124–131.

[44] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[45] LLNL Sequoia Benchmarks. (2014). [Online]. Available: https://codesign.llnl.gov/proxy-apps.php
[46] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Process. Lett.*, vol. 10, no. 02n03, pp. 215–226, 2000.



Chao Yu received the BE degree from the School of Information Engineering, Zhengzhou University, China, in 2012, and the ME degree from the School of Computer Science and Engineering, University of Electronic Science and Technology of China, in 2015. He is currently working toward the PhD degree in the School of Computer Science and Engineering, Beihang University. His research interests including parallel and distributed computing, high-performance computing, performance analysis.

Probir Roy received the BS degree from Bangladesh University of Engineering and Technology, Bangladesh, in 2009. He is currently working toward the PhD degree in Computer Science Department of The College of William and Mary. His research interests include parallel computing, compiler techniques, performance analysis and systems

Yuebin Bai received the PhD degree in computer science from Xian Jiao-tong University, Xian, China, in 2001. In 2003, he joined the faculty of Beihang University, where he is currently a full professor with the School of Computer Science and Engineering. His current research interests include computing system virtualization, real time and distributed systems, wireless networks. He is a senior member of the China Computer Federation (CCF), a member of the ACM and IEEE, and also a member of the IEICE.

Hailong Yang received the PhD degree from the School of Computer Science and Engineering, Beihang University, China in 2014. He is an assistant professor with the School of Computer Science and Engineering, Beihang University since 2014. His research interests include parallel and distributed computing, HPC, performance optimization and energy efficiency. He is also a member of IEEE and China Computer Federation (CCF).

Xu Liu received the BS degree from the School of Software, Beihang University, China, in 2006, the MS degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2009, and the PhD degree from Rice University, in 2014, working with Dr. John Mellor-Crummey. He is an assistant professor with the Computer Science Department of The College of William and Mary since 2014. His research interests include parallel computing, compiler techniques, performance analysis and systems.