2017 IEEE International Parallel and Distributed Processing Symposium

DR-BW: Identifying Bandwidth Contention in NUMA Architectures with Supervised Learning

Hao Xu, Shasha Wen†
College of William and Mary
{hxu07,swen}@email.wm.edu

Alfredo Gimenez, Todd Gamblin
Lawrence Livermore National Laboratory
alfredo.gimenez@gmail.com, tgamblin@llnl.gov

Xu Liu
College of William and Mary
xl10@cs.wm.edu

Abstract*—Non-Uniform Memory Access (NUMA) architectures are widely used in mainstream multi-socket computer systems to scale memory bandwidth. Without a NUMA-aware design, programs can suffer from significant performance degradation due to inter-socket bandwidth contention. However, identifying bandwidth contention is challenging. Existing methods measure bandwidth consumption. However, consumption alone is insufficient to quantify bandwidth contention. Furthermore, existing methods diagnose bandwidth for the entire program execution, but lack the ability to associate bandwidth performance to the source code and data structures involved. To address these challenges, we propose DR-BW, a new tool based on machine learning to identify bandwidth contention in NUMA architectures and provide optimization guidance. DR-BW first trains a set of micro benchmarks and extracts useful features to identify bandwidth contention via a supervised machine learning model. Our experiments show that DR-BW achieves more than 96% accuracy. Second, DR-BW associates memory accesses that incur bandwidth contention with data objects, which provides intuitive guidance for optimization. Third, we apply DR-BW to a number of real benchmarks. Our optimization based on the insights obtained from DR-BW yields up to a 6.5× speedup in modern NUMA architectures.

***Keywords*-NUMA, bandwidth contention, performance analysis, machine learning**

I. Introduction

The number of CPU cores per node in High Performance Computing (HPC) systems has increased rapidly in recent years, but the main memory bandwidth has not scaled with such a speed. Thus, bandwidth contention across cores due to main memory accesses has become a critical bottleneck. To mitigate this contention, modern HPC systems adopt Non-Uniform Memory Access (NUMA) architectures to scale the bandwidth of main memory. Figure 1 shows a typical NUMA architecture, which integrates four fully interconnected sockets, each with its own memory attached. Each core can access local memory attached to itself or remote memory attached to other socket. Local accesses have much lower latency and higher bandwidth than remote accesses. While this design makes it conceptually possible for cores to operate on independent portions of memory in parallel, it is also possible for cores on different sockets to contend for available memory bandwidth. With careless software design, bandwidth contention can occur in any memory controller, so it is critical to identify the root causes of bandwidth contention in NUMA applications to address performance problems.

Shasha Wen and Hao Xu make equivalent contribution to this work.

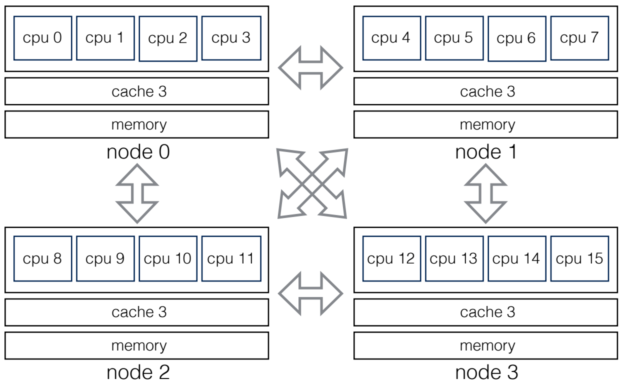


Fig. 1. An example NUMA architecture with four fully connected sockets.

Identifying problematic code that incurs bandwidth contention in NUMA architectures is challenging for several reasons. Runtime variation from hardware prefetching, parallel instruction pipelining, and operating system interference makes it difficult to predict contention via static analysis. While performance monitoring units (PMUs) are able to measure memory bandwidth consumption and count remote memory access requests, this information is insufficient to indicate whether bandwidth is suffering from contention. First, a high bandwidth consumption does not necessarily mean bandwidth contention. Furthermore, bandwidth contention can occur in any interconnect channel between sockets, so it is crucial to understand the NUMA topology to identify and localize contention.

Even if we can identify contention, deriving insights that would allow us to mitigate the problem remains challenging. Identifying contention indicates whether the problem exists, but without knowledge of the causes to the contention in an application, resolving the problem requires a substantial amount of domain knowledge and manual efforts. Existing tools such as HPCToolkit-NUMA [19] and MemProf [15] measure memory access latency to identify lines of code that cause problematic remote memory accesses, but neither addresses contention.

As a solution, we develop DR-BW, a lightweight profiler that automatically identifies bandwidth contention in NUMA architectures using supervised machine learning (ML) techniques. We make the following three contributions in DR-BW:

- DR-BW employs a supervised ML technique to train a highly accurate classifier for bandwidth contention. To the best of our knowledge, DR-BW is the first profiler that applies ML to diagnose bandwidth contention.
- DR-BW adopts a lightweight sampling mechanism available in modern CPU architectures. It collects all performance data to train our classifier in a single run, incurring less than 10% runtime overhead, on average.
- DR-BW not only identifies programs that suffer from memory contention, but also pinpoints problematic data structures used in the code. This insight provides straightforward guidance to optimize bandwidth contention.

DR-BW works on fully optimized binary code that runs at large scale on modern NUMA machines. It does not require any hardware or OS extensions. We train DR-BW with a set of micro benchmarks and apply the trained DR-BW to a set of real benchmarks from the Sequoia [17], Rodinia [6], NPB [3], and PARSEC [4] suites. DR-BW correctly detects bandwidth contention with 95% accuracy. Guided by DR-BW, we are able to optimize the code and achieve a up to 6.5× speedup in modern NUMA architectures.

DR-BW and the related benchmarks used in this paper are open sourced at https://github.com/xuhao417347761/DR-BW.

The rest of this paper is organized as follows. Section II reviews the existing work and distinguishes DR-BW. Section III overviews DR-BW and highlights the challenges in its design. Section IV describes the data collection and attribution used by DR-BW. Section V elaborates on the methodology for training DR-BW's classifier. Section VI explains the metrics used in DR-BW's diagnoser. Section VII and VIII evaluate DR-BW and show case studies. Finally, Section IX presents some conclusions and discusses future work.

II. Related Work

In this section, we review previous work on diagnosing problematic memory bandwidth performance and optimizing programs on NUMA architectures.

A. Memory Bandwidth Measurement

A number of existing tools such as HPCToolkit [2], VTune [11], and Perf [12] collect off-chip memory requests from hardware performance counters to quantify bandwidth consumption. However, the bandwidth consumption does not tell whether contention exists or not. For example, a regular pattern with high bandwidth consumption may not cause any bandwidth contention, while a random pattern with a low bandwidth consumption may incur intensive contention.

Instead of directly measuring bandwidth usage, Eklov et al. developed Bandwidth Bandit [10] to empirically measure a program's susceptibility to bandwidth contention problems. Bandwidth Bandit creates interference threads, which can be tuned to consume different amount of memory bandwidth. Because the available bandwidth is reduced, the monitored program may suffer a performance slowdown. If the interference thread incurs a large slowdown, the monitored program is bandwidth bound and subject to contention. Otherwise, the monitored program is not bottlenecked on bandwidth. Casas and Bronevetsky applied a similar approach to parallel programs [5].

The approach of utilizing interference threads is beneficial in determining whether a program's performance is sensitive to bandwidth contention but does not actually detect the occurrence of contention in an unmodified program. Furthermore, interference threads need to run on spare cores, but many parallel programs, especially HPC applications, use up all the cores available in the machine. In addition, this approach limits the analysis on the entire program level, lacking performance insights in fine-grained program contexts or semantics.

B. Heuristics for Bandwidth Contention

There are a number of tools that use heuristics to identify bandwidth contention issues and perform optimizations. Such tools exist within compilers [24], runtime systems [22], or operating systems [8], [7]. One approach determines bandwidth contention based on whether data allocated in one NUMA socket is accessed from threads in all sockets [20]. While effective for many workloads, this heuristic may not hold if the hardware pre-fetcher loads data into local caches in advance or if accesses from multiple sockets do not overlap in time. Other approaches use memory access latency as a heuristic—accesses that exceed a certain latency threshold are classified as contentious [7]. However, access latency varies due to a number of factors, and may not be indicative of contention in particular. In addition, determining an adequate threshold is usually difficult; some tools [19] determine its value via simple experiments.

Because no performance monitoring units currently exist to quantify bandwidth contention, using heuristics based on related measurements has proven feasible, but all aforementioned approaches are limited to the domains where the heuristics hold true. DR-BW builds upon the idea of heuristic-based detection, but instead of employing a single predefined heuristic, DR-BW adapts a statistical model for bandwidth contention by employing machine learning techniques on related performance measurements. Thus, DR-BW overcomes many of the limitations in previous work.

C. Machine Learning in Performance Analysis

Recent work has applied supervised learning to HPC performance bottleneck analysis. Sanath et al [13] use machine learning to detect false sharing and inefficient memory access. By training a classifier on a set of micro benchmarks with understood behaviors, they are able to detect the presence of false sharing in an application execution. ElMoustapha et al [23] build model trees to analyze the architecture performance. They aim to identify performance problems and estimate potential gain by addressing a specific performance issue. Wucherl Yoo builds ADP [27], an automated system to model, detect, and provide optimization suggestions for known

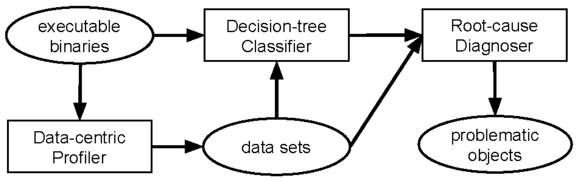


Fig. 2. Overview DR-BW's workflow for bandwidth contention detection and diagnosis.

performance "pathologies". ADP collects several hardware events for all functions in an application and uses them as inputs to a decision tree to classify each function according to a set of known pathologies. Vetter [26] uses machine learning to classify the communication inefficiencies in distributed application. Instead of overall analysis, Vetter detects each individual communication operations to see if it is efficient or not and reveals the cause of inefficiencies.

III. DR-BW Methodology and Overview

DR-BW addresses three major challenges in identifying bandwidth contention from hardware, software, and tool's views.

a) Bandwidth contention in complex interconnect channels: From the hardware perspective, the interconnect bandwidths between sockets vary, even for channels in opposing directions between sockets [18]. Understanding contention in specific interconnects and directions is important for guiding optimization. DR-BW monitors data transmission in each channel and associates contention with specific channels.

b) Root-cause analysis: From the software perspective, problematic memory accesses can be buried deep in complex codebases. Moreover, understanding the problematic access instructions alone does not lead to straightforward optimization strategies. DR-BW applies root-cause analysis to associate problematic accesses with data objects. Accesses to these data objects can then be optimized by modifying their allocation schemes for different NUMA architectures.

c) Bandwidth contention mini programs: Another challenge is collecting meaningful data on which to train the classifier. No standard benchmark suite exists for bandwidth contention, so we developed a set of problem-specific mini programs, each tunable to run with different configurations.

Figure 2 shows an overview of DR-BW. The profiler monitors the execution of fully optimized binary code and collects performance data. The performance data is fed into a decision-tree classifier to determine whether contention has occurred in the program execution or not. If DR-BW identifies contention in a program, its diagnoser further analyzes the code to identify the root cause of the contention, including accesses to problematic data objects. DR-BW associates the analysis with source code to provide intuitive optimization guidance. In the remaining paper, Section IV describes the design of DR-BW's profiler, Section V elaborates on the design of DR-BW's classifier and Section VI explains the metrics used in DR-BW's diagnoser.

IV. DR-BW's Profiler

The profiler component in DR-BW collects memory accesses and extracts performance features that are used as input to the contention training and detection phases. To guarantee lightweight analysis, DR-BW relies on low-overhead hardware performance monitoring units (PMUs) to do the measurement. PMUs collect a variety of statistics during the execution, quantify some performance metrics, and help to identify potential bottlenecks. In addition, to accurately predict the bandwidth for complex interconnection topologies and diagnose the root cause of the contention, there are two extra features implemented in the profiler. One is to distribute the samples to corresponding channels; the other is to associate the samples with allocated data objects in the program.

A. Address Sampling

In order to provide DR-BW with sufficient information for measurement, prediction and diagnosis, the PMU sampling we use should:

- Report the effective memory address that is read or written by the sample, and the memory layer this sample is touching: L1, L2, L3, local DRAM or remote DRAM.
- Collect memory related metrics along with the sample. Such metrics include local/remote NUMA memory accesses, cache misses, or latency.
- Record the CPU ID where each sampled memory access instruction executes.

Address sampling supported in modern PMUs can meet all of these requirements. Examples include Intel's precise event-based sampling (PEBS) with latency extensions [1] (supported on Nehalem, SandyBridge, IvyBridge, Haswell, and Broadwell microarchitectures); AMD Opteron processors with instruction-based sampling for micro ops (IBS op) [9]; and IBM POWER5 and later generations of POWER architectures that count marked events (MRK) [25]. In this paper, we conduct all our experiments is an Intel SandyBridge machine. We use PEBS PMUs to sample memory event `MEM_TRANS_RETIRED:LATENCY_ABOVE_THRESHOLD`. We sample one of every 2000 memory accesses independently in each thread. DR-BW uses PEBS to collect memory samples and report metrics. We will extend our DR-BW on AMD and IBM platforms in future work.

B. Associate Samples with Channels

A single memory sample can be located on the channel from any NUMA node to another. We assume that bandwidth issues on one channel are mainly identified by accesses on that channel. For example, we use only samples observed between nodes 0 and 1 to diagnose performance problems on the bus connecting nodes 0 and 1 (not samples that occurred between nodes 0 and 2). Instead of predicting problems for the entire execution, DR-BW detects bandwidth issues per-channel.

Thus, it is necessary to associate all the samples with channels based on their sources and targets. The source of a sample, also known as the `accessing node`, is the node where the processor triggers the memory access. With the precise CPU ID and the hardware topology, the NUMA node that a core resides is easy to obtain. The target of a sample is the `locating node`, where the data reside. To find the target, we use the `libnuma` library to get the location node with the precise memory address reported by this sample. With the sources and targets, the samples are then associated with different channels.

C. Attribute Samples to Data Objects

After sample distribution, DR-BW can use the samples from one channel to predict if there would be contention on that channel. When contention is predicted, to better understand the root cause of the contention, DR-BW attributes the samples to the data objects.

There can be three different data types in a program, static data, stack data, and heap allocated data. Compared with static and stack ones, the dynamically allocated data usually have larger sizes and suffer more from the bandwidth contention. So in the current implementation of DR-BW's profiler, we focus on the heap allocated data. To build connections between data objects and the samples, DR-BW's profiler intercepts all the heap allocations (malloc family functions, such as `malloc`, `calloc`, `realloc`). For each allocation point, the profiler maintains an entry in a table to record the instruction pointer of the allocation and the allocated memory ranges. Later on, when an address sample is triggered, by comparing its memory address with the memory ranges recorded in the table, we can associate this sample with the corresponding data object.

V. DR-BW's CLASSIFIER

The decision-tree classification algorithm is widely used in predicting the answer to a yes/no question. DR-BW uses a decision tree to answer whether there would be bandwidth contention for a program. To build and train this decision tree, we need a set of benchmarks which cover both bandwidth contention and non-contention scenarios. There does not exist a standard benchmark suite for this purpose, so we develop several micro benchmarks and tune the data sizes to run these benchmarks in either bandwidth friendly mode or contention mode. In this section, we describe how we build our micro benchmarks and how we train the decision-tree model.

A. Mini-programs for Training

1) Multithreaded vector operations: The first set of programs is OpenMP multithreaded vector operations. The summary of these programs is as follows:

- vector summation (sumv): each thread computes the summation of its own share of vector data.
- dot-product of vectors (dotv): each thread computes the dot-product of its own share of vector data.
- count for vectors (countv): each thread counts the number of occurrences of a specific number in its own share of vector data.

We use the name `sumv, dotv, countv` to refer to these three programs in later sections. `sumv, dotv` and `countv` differ in memory usage and memory access pattern. The size of the vector is tuned to adjust the bandwidth friendliness. As the size input data grows, cache misses and remote accesses increase. When the execution time of these programs do not grow proportional to the input sizes, we believe contention in remote bandwidth occurs. This is because the contention can largely delay the memory requests and incurs significantly longer execution time.

2) Single-threaded bandit program: We design the bandit program to continuously issue memory requests without cache hits. We use the bandit program to study main memory bandwidth and avoid the interference by caches. The implementation of the bandit program follows an existing approach [10]. The bandit program issues every memory access that conflicts with its previous one in caches so the request goes to the main memory. To achieve this, we first allocate several huge pages to ensure that we can have a deterministic mapping between the page offset and cache set. We then form a stream of memory accesses with a pointer chasing pattern. All memory accesses touch addresses that are mapped to the same cache sets, causing conflict cache misses. By placing huge pages in remote memory, we are able to evaluate the remote memory bandwidth. In the training phase, we tune the number of streams in one bandit instance and the number of co-running bandit instances to ensure different requirements to memory bandwidth.

To simplify the description of whether one run of a program suffers from bandwidth contention, we define the following two modes for each running instance:

- **good**: i.e, no remote memory bandwidth contention
- **rmc**: with remote memory bandwidth contention

When one application running under a configuration has contention, we say this run results in the "`rmc`" mode.

B. Identification of Performance Features

To predict the memory contention, DR-BW mainly measures information that are memory related. Given one memory access, all the features related to it can be classified into the following three categories:

- **Identification** means features that can be used to identify the memory access, including the `memory address` and the `source node, CPU id, thread id` that triggered this access.
- **Location** includes features specifying where this memory is located and which layer of the memory hierarchy this access is touching. Information like `L1 Hit, L2 Hit, L3 Hit, L3 Miss, DRAM access, Remote DRAM access` belong to this category.
- **Latency**, or how many CPU cycles it takes to complete an access.

DR-BW records all these features for each memory sample, when a batch of memory samples are collected, the categories above can be derived to further statistics features:

- **Statistics Identification** pulls in features like `number of memory accesses` triggered by `CPU id, thread id, node id`.

TABLE I
SELECTED FEATURES TO TRAIN DR-BW'S CLASSIFIER.

Feature	Feature Description
1	Ratio of latency above 1000 among all samples
2	Ratio of latency above 500 among all samples
3	Ratio of latency above 200 among all samples
4	Ratio of latency above 100 among all samples
5	Ratio of latency above 50 among all samples
6	# of remote dram access sample
7	Average remote dram access latency
8	# of local dram access sample
9	Average local dram access latency
10	Total # of memory access sample
11	Average memory access latency
12	Total # of line fill buffer access sample
13	Line fill buffer access latency

TABLE II
SUMMARY OF THE COLLECTED TRAINING DATA.

mini-programs	good	rmc	Total
sumv	24	24	48
dotv	24	24	48
countv	24	24	48
bandit	48	-	48
Full training data set	**120**	**72**	**192**

- **Statistics Location** includes features saying the total number happened in one memory layer, such as `Num_L1_Hit`, `Num_L2_Hit`, `Num_L3_Hit`, `Num_L3_Miss`, `Num_DRAM_access`, `Num_RemoteDRAM_access`.
- **Statistics Latency** adds features quantifying the `ratios` of different latency among the samples and `Average latency` of memory accesses across different memory layers.

All these statistics features are included in a candidate list for training and prediction. We call it candidate list because it is impractical and unnecessary to use all the features in the prediction. It is reasonable to select and use the features that are highly relevant to bandwidth contention.

In the selection phase, each of our multi-threaded mini-programs is executed in both `"good"` and `"rmc"` modes, with different thread numbers (e.g. 1, 2, 4, 8, and 16 in a NUMA node). We measure each candidate feature. If there is significant difference in the statistics between `"good"` and `"rmc"` for a majority of mini-programs, this candidate feature is selected as a relevant one for the future prediction.

In the experiments, we notice that there are some remote memory events which we thought should be related with contention, but are actually not, such as `Mem_Load_Uops_LLC_Miss_Retired.Remote_DRAM`. Table I shows the selected features DR-BW uses.

C. Collection of Training Data

We use decision tree classification algorithm in Statistics and Machine learning toolbox of Matlab 2016a. The training process is conducted on the platform described in Section VII.

Training datasets are the selected statistics collected by running the mini-programs. Each mini-program is run with multiple configurations. The configuration includes a set of problem size, number of threads and threads to nodes binding. Each configuration is either in `"good"` mode or `"rmc"` mode. Then we collected all initialized data, and manually examined each of them. The result of data collection is summarized in table II. Our overall training data set, has 192 instances. We manually label each training data instance by adding corresponding mode(`"good"`,`"rmc"`) as separate field.

TABLE III
CONFUSION MATRIX FOR THE TRAINING DATA.

		Predicted Class	
		good	rmc
Actual Class	good	118	2
	rmc	3	69

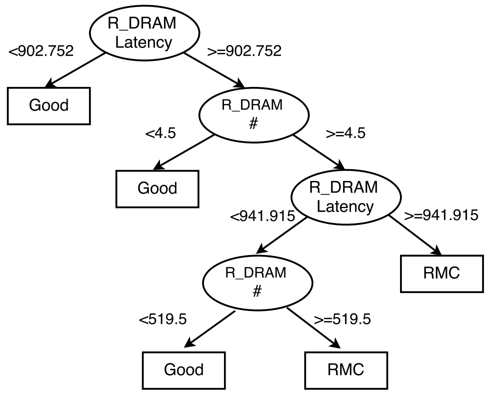


Fig. 3. The decision tree used by DR-BW. The internal nodes are labeled with "features", while the leaf nodes are labeled with "classifications".

D. Classifier of the Decision Tree

Figure 3 shows the decision tree generated with the training datasets. The model uses two features (features numbered with 6 and 7 in Table I). In every internal node in the tree, branching is to the right if the normalized value of the corresponding feature is above a threshold and otherwise, to the left. To verify the effectiveness of the decision-tree algorithm, we have applied stratified 10-fold cross validation on the training data. It shows a 187/192 (or 97.4%) overall success rate. Table III shows the confusion matrix. Misclassification sometimes occurs because DR-BW depends on hardware sampling, which does not monitor every memory access.

VI. DR-BW'S DIAGNOSER

Once DR-BW's classifier detects that the application does have a bandwidth contention issue, we apply DR-BW's root-cause diagnoser to further identify why this contention happens. In the diagnoser, we develop metrics to quantify the contribution to the contention of all the data objects. The data which incur most to the contention requires further investigation.

A. Quantify Data Object's Contribution to Contention

In DR-BW's profiler component, we tagged each sample with an allocation point, so we know exactly which data object each sample is touching. On the other hand, in DR-BW's classifier, we detect the contention issue for each channel. So we know which samples in which channel cause the contention. To quantify data's contribution, we can quantify how these samples are distributed among the data objects.

a) Metrics per channel: For a channel c connecting two NUMA nodes, which has a contention issue, all the samples in c are aggregate based on the data objects they are touching. We define *Contribution Fraction (CF)* for a data object A in channel c as follows:

$$CF_c(A) = \frac{Samples(c, A)}{Samples(c, ALL)}$$

$Samples(c, A)$ means the total number of samples accessing data A in channel c while $Samples(c, ALL)$ is the total number of samples happen in this channel. The *Contribution Fraction* for one data object specifies its contribution to the contention in that channel.

b) Metrics cross channels: When accumulating the contribution across channels, we count all the samples in the channels that have contention issues. For those channels that do not have any contention issue, we do not further analyze their samples. Thus, the *Contribution Fraction (CF)* for data A over all the channels involved in the program is:

$$CF(A) = \frac{\sum_{c=0}^{N} Samples(c, A)}{\sum_{c=0}^{N} Samples(c, ALL)}$$

N is the total number of contented channels. The sum of *CF* for all the data objects used in the program should be 1.

B. Root-cause Blaming

After we get the *CF* for all the data used in one program, we can rank the data objects based on the *CF* values. The data objects with the highest *CF* are the root causes of bandwidth contention. To alleviate the contention, one should applying optimization methods, such as collocating the data with their computation to the top data objects identified by DR-BW.

VII. Evaluation of the Decision-Tree Classifier

In this section we evaluate our decision-tree classifier with real world benchmarks from the following benchmark suites:

- **NAS Parallel Benchmarks (NPB)** [3] are a small set of programs derived from computational fluid dynamics applications. It includes benchmarks for instructed adaptive mesh, parallel I/O, multi-zone applications and computational grids.
- **PARSEC** [4], short for the Princeton Application Repository for Shared-Memory Computers, contains multi-threaded programs focusing on emerging workloads.
- **Rodinia** [6] is a parallel benchmark suite containing computation-intensive application with diverse accelerators. Paralleled codes are provided with different engines. We run with the OpenMP ones.

TABLE IV
Benchmark classification.

Class	Benchmarks
good	BT, CG, DC, EP, FT, IS, LU, MG, UA
	Blackscholes, Bodytrack, Ferret, Fluidanimate, Freqmine, Raytrace, Swaptions, X264
rmc	SP
	Streamcluster
	Needleman_Wunsch
	AMG2006, IRSmk, LULESH

- **Sequoia** [17] is a benchmark suite published by Lawrence Livermore National Laboratory. Memory access patterns in these benchmarks are highly representative.
- **LULESH** [16] also developed by LLNL solves the Sedov blast wave problem for one material in 3D. We use the OpenMP parallelized version with C++ codes.

Our experiment platform is a 32-core (8 cores $\times$ 4 sockets) Intel Xeon CPU E5-4650 machine clocked at 2.70GHz. The machine has 32KB L1 cache, 256KB L2 cache per core, 20MB L3 cache per socket and 256GB (64GB $\times$ 4 sockets) DRAM. All the benchmarks are compiled with `gcc 4.8.5 -O3`.

A. Benchmark Classification Results

We applied our classifier model on 23 benchmarks selected from the benchmark suites above. Each program is run with different combinations of input sets, number of threads, and NUMA nodes.

We run PARSEC benchmarks with four input sets: `native`, `simLarge`, `simMedium` and `simSmall`. NPB benchmarks are run with CLASS `A`, `B` and `C`. For Rodinia and Sequoia benchmarks, we run with the provided default input size and also tune the parameters to make it both smaller and larger.

We use `Tt-Nn` to represent a specific configuration with total t threads and n nodes used. The total t threads are evenly distributed among the n nodes. Threads are also bound to the cores, e.g. for `T16-N4` configuration, `threads 0-3` are bound to `node 0`, `threads 4-7` are in `node 1`, `threads 8-11` are in `node 2`, and `threads 12-15` are in `node 3`. Our experimental platform has 4-node and 32-core with Hyper-Threading Technology. We tuned t to be 16, 24, 32 and 64 and n to be 2, 3, 4. For each node, we have t/n threads assigned. In total, we have eight configurations (`T16-N4`, `T24-N4`, `T32-N4`, `T64-N4`, `T24-N3`, `T16-N2`, `T24-N2` ,`T32-N2`).

We applied our classifier on each channel and use the following rules to classify the detection result:

1) For a specific case of a benchmark (case here denotes specific inputs, specific threads and NUMA nodes affinity), if there is at least one remote access channel which is detected to have contention, we treat this case as `"rmc"`. Otherwise, it will be treated as `"good"` .
2) For a benchmark program with all different cases, if there is at least one of them has remote memory contention issue, we treat this program as `"rmc"`. Otherwise it will be treated as `"good"`.

TABLE V
EVALUATING THE ACCURACY OF DR-BW WITH ALL THE REAL BENCHMARKS.

Benchmark	**# cases**	**Actual**		**Detected**	
		RMC	**NO RMC**	**RMC**	**NO RMC**
Swaptions	32	0	32	0	32
Blackscholes	32	0	32	0	32
Bodytrack	16	0	16	0	16
Freqmine	32	0	32	0	32
Ferret	32	0	32	0	32
Fluidanimate	32	0	32	4	28
X264	32	0	32	0	32
Streamcluster	**16**	**13**	**3**	**16**	**0**
IRSmk	**24**	**15**	**9**	**15**	**9**
AMG2006	**8**	**8**	**0**	**8**	**0**
NW	**24**	**16**	**8**	**17**	**7**
BT	24	0	24	0	24
CG	24	0	24	0	24
DC	16	0	16	0	16
EP	24	0	24	0	24
FT	24	0	24	2	22
IS	24	0	24	0	24
LU	24	0	24	0	24
MG	24	0	24	0	24
UA	24	0	24	9	15
SP	**24**	**11**	**13**	**11**	**13**
Total (Overall)	512	63	449	82	430

Table IV shows the classification summary of the 23 benchmarks. The classification is the overall result considering all different cases. 17 benchmarks are classified in the `"good"` class as they do not show remote bandwidth contention issue with all the input size and threads we tried. We got six programs with remote memory contention issues, so contention happens at least during one configuration run. We will discuss the results of them in detail in the following subsections.

B. Classification Statistics

It is not straightforward to test how accurate our classifier is over the real benchmarks because there is no prior-knowledge on the existence of bandwidth contention in these benchmarks. Moreover, the contention varies according to different hardware parameters and execution configurations. To address this issue, we build our evaluation based on a assumption that remote bandwidth contention will benefit from the memory interleaving. Because memory interleaving is able to balance the memory requests across different NUMA domain, it alleviate memory bandwidth contention. Thus, if the speedup of the interleaved version exceeds a predefined threshold 10% over the original code, we believe this benchmark suffers from a contention issue. We treat the results obtained from this method as the ground truth.

Tables V shows the summary of the benchmarks detected with our classifier methodology and the interleaved classification method. This table shows the total number of instances we run for each benchmark with different combination of input, threads and nodes. Among those instances, we identify how many of them are detected to have bandwidth contention and how many of them are contention free. The `Actual` columns show the result of interleaved classification method (ground truth) while the `Detected` columns show the result of our decision-tree classifier. We can see that for most cases, the two classification methods show the same results, which highlights the accuracy of DR-BW.

TABLE VI
QUANTIFYING DR-BW'S ACCURACY WHEN ANALYZING REMOTE MEMORY BANDWIDTH CONTENTION.

		Detection Classification	
		RMC	**No RMC**
Actual	**RMC**	63	0
	No RMC	19	430

Correctness	(430+63)/(0+63+19+430) = 96.3%
False positive Rate	19/(19+430)=4.2%
False negative Rate	0/(0+63)=0%

TABLE VII
DR-BW'S RUNTIME OVERHEAD.

Code	Execution time (s)		Overhead (%)
	Without profiling	With profiling	
IRSmk	118.1	119.2	+0.9
AMG2006	122.5	132.1	+7.8
Streamcluster	245.2	222.6	-9.2
NW	55.8	59.4	+6.4
SP	411.0	425.7	+3.6
LULESH	118.2	130.0	+10.0
Average	-	-	+3.3

We further evaluate our detection method in Table VI. Compared to "actual" in all Benchmarks, We have been able to detect remote memory contention with no false negative and 96.3% overall correctness. Thus, we can infer that DR-BW successfully detects remote memory contention problems in `Streamcluster`, `AMG2006`, `IRSmk`, `SP` and `NW`.

VIII. CASE STUDIES

Among the total 23 benchmarks we investigate, DR-BW's classifier detects six suffering from the remote bandwidth contention issue. In this section, we further study these benchmarks with DR-BW's root-cause diagnoser, pinpoint and optimize the problematic data objects.

The profiling overhead when using all the 64 cores across four NUMA nodes for these benchmarks is shown in Table VII. Time is averaged after four executions. As the table shows, the highest overhead we have is for `LULESH`, which is 10.0%. The average overhead of the six benchmarks is 3.3%. Particularly, `Streamcluster` runs 9% faster with profiling, because, to the best of our knowledge, the profiling code interferences the original memory accesses and reduces the bandwidth contention. In the following subsections, we discuss each benchmark one by one.

A. AMG2006

AMG2006, one of LLNL Sequoia benchmarks, is a parallel algebraic multi-grid solver for linear systems arising from problems on unstructured grids. It consists of three phases: initialization, setup and solver. It is written in C with MPI and OpenMP.

With DR-BW's diagnoser analysis, we calculate the *CF* for all the data objects used in AMG2006. Figure 4(a) shows the distribution of *CF* among the data objects when running with

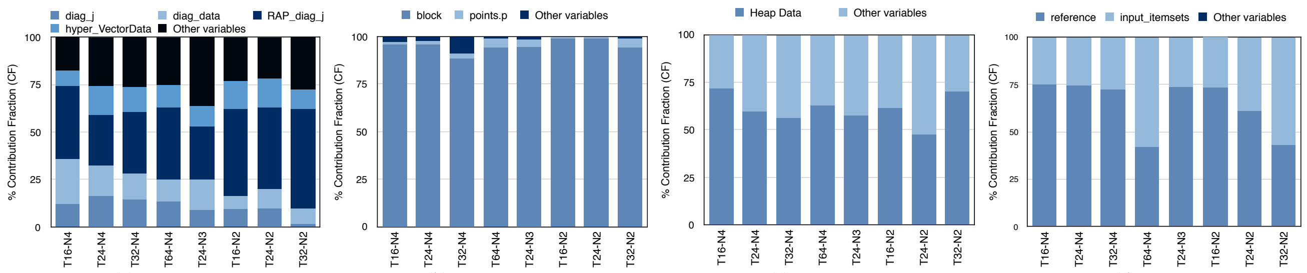

(a) AMG2006 (b) Streamcluster (c) LULESH (d) NW

Fig. 4. *Contribution Fraction (CF)* distribution across data objects in different benchmarks.

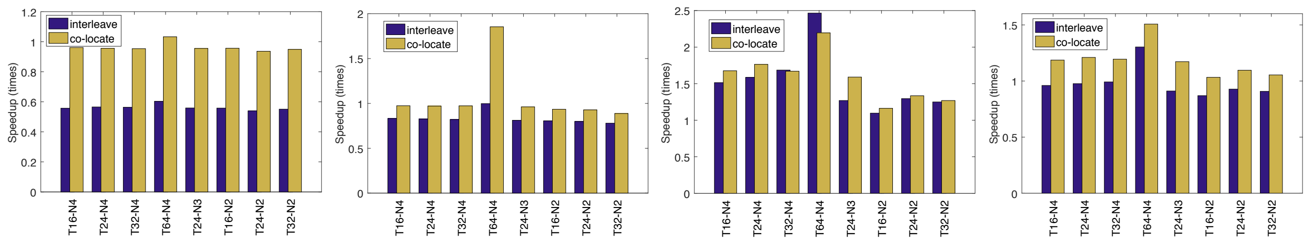

(a) Initialization (b) Setup (c) Solver (d) Total

Fig. 5. Speedups in different AMG2006 phases with different execution configurations after our optimization.

30 × 30 × 30 grid size. Four of them, having high *CF* are highly related with the contention when running with different numbers of threads pinned to different number of nodes.

Among the four data objects listed, array `RAP_diag_j` is the most highly related with the contention no matter how many threads and nodes are set to run the program. Moreover, arrays `diag_j` and `diag_data`'s contention contribution grows when more NUMA nodes are used for computation. We then manually check how these four arrays are used in the program. We find that they are used in an OpenMP parallel `for` loop, and each thread handles a continuous segment of the array. To optimize the code, we break the data into multiple segments and co-locate each with its computation at the array allocation point. We use `libnuma` [14] to control the memory allocation.

We optimize with data-computation collocation for all the four data objects as shown in Figure 4(a); Figure 5 demonstrates the speedup of our optimization with different execution parameters at different execution phases. We also compare this speedup with the memory interleaved optimization, which interleaves all the memory pages allocated for the entire program. We can see that our optimization that focuses on data objected identified by DR-BW achieves higher speedups. As shown in Figure 5, the *interleave* optimization achieves good performance (1.5× in average) in the solver phase, but this coarse-grained optimization hurts the setup and initialization phases. Our *co-locate* optimization guided by DR-BW achieves the same high speedup in the solver phase without hurting the setup and initialization phases. Thus, our optimization has higher speedups for the entire program execution over the *interleave* optimization.

We profile the optimized code with 64 threads in four NUMA nodes. The total number of remote memory accesses is reduced by 87.8% and the average memory access latency is decreased by 83%.

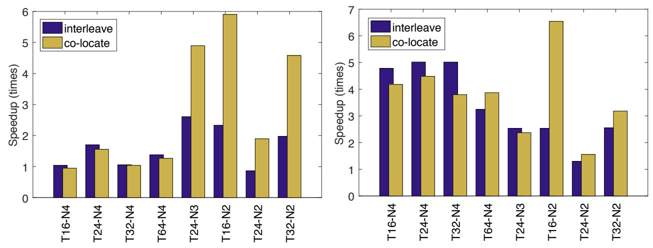
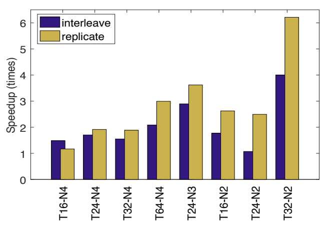

(a) Medium (64 works) (b) Large (96 works)

Fig. 6. Speedups of IRSmk with different input sizes and execution configurations.

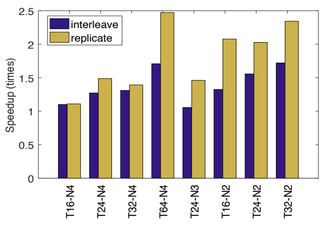
B. IRSmk

IRSmk, also from LLNL Sequoia benchmarks, is a parallel Implicit Radiation Solver for diffusion equation on a three-dimensional, block structured mesh. It is implemented in C with OpenMP. We evaluate a highly optimized version of IRSmk obtained from previous work [21]. DR-BW's diagnoser detects 29 problematic arrays including array `b`, `k` and other 27 arrays, which are of the same size and show similar access patterns. These arrays share similar *CF* values and uniformly contribute to the bandwidth contention.

To optimize the code, we apply the *co-locate* optimization for all the 29 variables. And we test our speedup along with the *interleave* optimization with different input sizes, nodes and threads as shown in Figure 6. Medium and large mean that we run with 64×64×64 and 96×96×96 input meshes,



(a) simLarge



(b) Native

Fig. 7. Speedups for Streamcluster with different input sizes and execution configurations.

respectively.

When the input size is smaller, e.g., with the configuration of `T16-N4`, both optimization strategies do not show significant speedups. However, with the growing of input sizes, the benefit of *co-locate* and *interleave* policies becomes more significant. The maximum speedup can be as high as $6.2\times$. With all four NUMA nodes utilized and the number of threads bound to each NUMA node less than eight, *interleave* can have slightly shorter execution time than *co-locate*. However, *co-locate* performs much better when fewer NUMA nodes are used. When running large input with 64 threads and 4 NUMA nodes, the total remote memory accesses is reduced by 72.5% and the average memory access latency is decreased by 88.9%.

C. Streamcluster

DR-BW identifies that `Streamcluster` from PARSEC has a remote memory bandwidth contention issue, which, actually, has been verified in previous work [7]. We evaluate `Streamcluster` with two different input sizes, `native` and `simLarge`. Figure 4(b) shows problematic data structures identified by DR-BW's diagnoser. When running with the native input, two arrays (`block` and `point.p`) account for more than 90% of the contention. Among them, the array `block` is more important since it has the highest *CF* value.

Our further analysis shows that `block` is randomly accessed by all the threads and the data is never overwritten after the initialization. Thus, we create a shadow replications of `block` for the threads in each NUMA node, so all the accesses to `block` can go to local memory. Figure 7 shows the speedups with the *replicate* optimization as well as the *interleave* optimization when running with different inputs. We can see that when running with larger number of nodes, e.g. three or four, the *interleave* and *replicate* show similar improvement over the original execution. However, when fewer nodes and threads are used, *replicate* performs much better. This is because the *interleave* optimization balances the bandwidth requests, but it also introduces more remote memory accesses. When running with only two nodes and a small number of threads, the bandwidth contention is not that serious, so the overhead of increased remote accesses surpasses the benefit of bandwidth contention reduction.

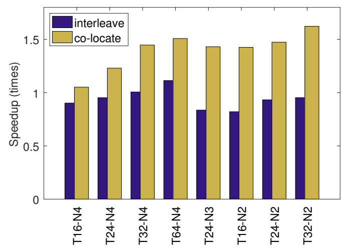


Fig. 8. Speedups for LULESH with different execution configurations.

D. LULESH

We evaluate `LULESH` with one large input size. In `LULESH`, there are over 40 heap allocated arrays, which show similar sizes and accessing patterns. As shown in Figure 4(c), heap data objects allocated at `line:2158-2238` account for a sum of the *CF* higher than 50%. There are some other intractable data objects that account for non-negligible *CF*. Particularly, two static data objects used in `LULESH` incur significant memory accesses. DR-BW currently does not support tracing static data object, so we leave it as our further work.

To optimize the heap allocated data, similarly as `IRSmk`, we co-locate the data with the computations. Figure 8 shows execution time speedups of our *co-locate* and also *interleave*. We can clearly see that, the speedup of *co-locate* performs much better than *interleave*. When running with `T16-N4` configuration, there is not significant speedup, since the DR-BW's classifier puts the execution with this configuration in `"good"` category. This is because only four threads bound to each NUMA node are not enough to saturate remote memory bandwidths.

The optimized code reduces the number of remote accessed by 50% and the average latency by 67%, when running with 64 threads across four NUMA nodes.

E. Rodinia NW

In Rodinia `NW` , DR-BW pinpoints two problematic data structures `reference` and `input_itemsets` with *CF* values shown in Figure 4(d). Both arrays are allocated by the master thread but accessed by threads across all NUMA nodes. To address this problem, we co-locate the allocation of these two arrays with computations across all NUMA nodes by using `libnuma`. This achieves a speeded up of 32.6%. After the optimization, the average memory access latency is reduced by 60%.

F. SP

DR-BW's classifier detects that SP with `C` class input size has a remote memory bandwidth contention issue. All the data objects used in this project are global data which are statically allocated. As we do not track the static data, we simply quantify the execution speedups obtained with the *interleave* optimization. When the threads per node is high (e.g., > 8), the speedup is as high as $1.75\times$, when running with 64 threads across four NUMA nodes.

G. Blackscholes

To evaluate DR-BW, we also choose benchmarks which fall into the "good" category and apply optimizations to see whether we can obtain performance benefit. The study of `Blackscholes` from PARSEC is for this purpose. DR-BW's classifier reports that `Blackscholes` with native input size has no remote memory bandwidth contention issue. We evaluate the execution time of *interleave* optimization with all configurations. The difference with the original execution is negligible. With further analysis, DR-BW highlights the array `buffer` associated with the highest CF score. We optimize the `buffer` by collocating the data with computations, but the speedup is $< 1\%$. Thus, DR-BW successfully shows that `Blackscholes` does not incur bandwidth contention, which applies to other benchmarks in the "good" category.

IX. Conclusions and Future work

In this paper, we present the design and implementation of DR-BW, a profiler that uses machine learning techniques to identify bandwidth contention in NUMA architectures. DR-BW collects performance data with low overhead, feeds the data into a novel machine learning model to identify contention, and associates the analysis results with both programs and significant data objects. We study a number of benchmarks and show that DR-BW achieves more than 96% accuracy. With several case studies, we demonstrate that DR-BW is able to guide performance optimization and yield up to a 6.5× speedup. In the future, we will extend DR-BW to identify resource contention beyond memory bandwidth using machine learning techniques, such as contention in instruction issue slots, different level of caches, and I/O devices.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and the National Science Foundation (NSF) under Grant No. 1618620 and 1464157.

References

[1] Intel® 64 and ia-32 architectures software developers manual. 2010.

[2] L. Adhianto et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 2010.

[3] D. H. Bailey, E. Barszcz, et al. The NAS parallel benchmarks – summary and preliminary results. In *Proc. of the 1991 ACM/IEEE conference on Supercomputing*, 1991.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. of the 17th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2008.

[5] M. Casas and G. Bronevetsky. Active measurement of memory resource consumption. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[6] 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. of the 2009 IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2009.

[7] M. Dashti et al. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proc. of the* 18^{th} *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[8] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß. kmaf: Automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[9] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf, November 2007. Last accessed: Dec. 13, 2013.

[10] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth bandit: Quantitative characterization of memory contention. In *2013 IEEE/ACM International Symposium on Code Generation and Optimization*, 2013.

[11] Intel Corporation. Intel VTune performance analyzer. http://www.intel.com/software/products/vtune.

[12] Intel Corporation. Linux performance tool. http://www.brendangregg.com/linuxperf.html.

[13] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. De Silva, S. Rathnayake, X. Meng, and Y. Liu. Detection of false sharing using machine learning. In *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–9. IEEE, 2013.

[14] A. Kleen. A NUMA API for Linux. http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf, 2005. Last accessed: Dec. 12, 2013.

[15] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A memory profiler for NUMA multicore systems. In *Proc. of the 2012 USENIX Annual Technical Conf.*, Berkeley, CA, USA, 2012.

[16] Lawrence Livermore National Laboratory. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://codesign.llnl.gov/lulesh.php. Last accessed: Dec. 12, 2013.

[17] Lawrence Livermore National Laboratory. LLNL Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks. Last accessed: Dec. 12, 2013.

[18] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*.

[19] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2014.

[20] X. Liu and J. M. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proc. of the 2013 ACM/IEEE Conference on Supercomputing*, 2013.

[21] X. Liu, K. Sharma, and J. Mellor-Crummey. Arraytool: a lightweight profiler to guide array regrouping. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 405–416. ACM, 2014.

[22] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–99. ACM, 2006.

[23] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 116–125. IEEE, 2007.

[24] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler support for selective page migration in numa architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[25] M. Srinivas, B. Sinharoy, R. Eickemeyer, R. Raghavan, S. Kunkel, T. Chen, W. Maron, D. Flemming, A. Blanchard, and P. Seshadri. IBM POWER7 performance modeling, verification, and evaluation. *IBM Journal of Research and Development*, 55(3):4–1, 2011.

[26] J. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *Proceedings of the 14th international conference on Supercomputing*, pages 245–254. ACM, 2000.

[27] W. Yoo. *Automated performance characterization of applications using hardware monitoring events*. PhD thesis, University of Illinois at Urbana-Champaign, 2013.