# (Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads

Zoltan Majo and Thomas R. Gross
*Department of Computer Science*
*ETH Zurich, Switzerland*

***Abstract*—An important aspect of workload characterization is understanding memory system performance (i.e., understanding a workload's interaction with the memory system). On systems with a non-uniform memory architecture (NUMA) the performance critically depends on the distribution of data and computations. The actual memory access patterns have a large influence on performance on systems with aggressive prefetcher units. This paper describes an analysis of the memory system performance of multithreaded programs and shows that some programs are (unintentionally) structured so that they use the memory system of today's NUMA-multicores inefficiently: Programs exhibit program-level data sharing, a performance-limiting factor that makes data and computation distribution in NUMA systems difficult. Moreover, many programs have irregular memory access patterns that are hard to predict by processor prefetcher units.**

**The memory system performance as observed for a given program on a specific platform depends also on many algorithm and implementation decisions. The paper shows that a set of simple algorithmic changes coupled with commonly available OS functionality suffice to eliminate data sharing and to regularize the memory access patterns for a subset of the PARSEC parallel benchmarks. These simple source-level changes result in performance improvements of up to 3.1X, but more importantly, they lead to a fairer and more accurate performance evaluation on NUMA-multicore systems. They also illustrate the importance of carefully considering all details of algorithms and architectures to avoid drawing incorrect conclusions.**

## I. Introduction

As multicore architectures become widespread, parallel software becomes more and more important. Writing well-performing parallel programs is, however, far from easy, thus parallel programs often suffer from problems that limit their performance scalability.

There are many reasons for the lack of scalability of parallel programs. Possible reasons include both software- and hardware-related problems [14]. A crucial aspect, with a large influence on the scalability of parallel programs, is memory system performance: Parallel programs must use the memory system of the architecture efficiently to achieve good performance scalability. As a result, programmers must be able to diagnose and solve performance problems related to the memory system to be able to write well-performing parallel programs. This level of understanding requires programmers to appreciate the details of the memory system as well as the program's interaction with the memory system.

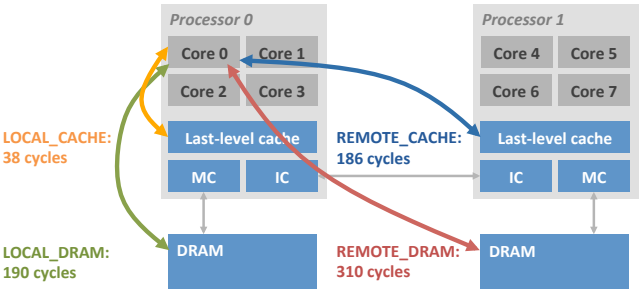



Figure 1: 2-processor NUMA-multicore system.

Understanding memory system performance is, however, far from trivial. Modern memory systems include many different mechanisms, all intended to provide good performance, but all of which interact with workloads in ways that are sometimes unexpected and whose effects can be difficult to fully understand. E.g., many recent microarchitectures include aggressive prefetcher units. Prefetcher units predict the memory access patterns of programs and fetch data into caches before the data is accessed by processor cores so that the performance impact of long-latency memory accesses is alleviated. Nevertheless, if a program has irregular memory access patterns, prefetcher units are ineffective, and the program experiences the full latency of its memory accesses.

Another aspect that makes understanding memory system performance difficult is the continuous evolution of memory system architectures. As the number of processor cores increases with (almost) every new processor generation, the design of the memory system is often revised so that the memory system satisfies the increased demand for memory bandwidth but memory access latencies are kept low at the same time. As a result, knowledge a programmer has about a particular memory system design does not necessarily apply to the next generation of microarchitectures. Moreover, new memory system designs can pose challenges non-existent on architectures of a previous generation.

An example of a recent evolution in memory system design is the non-uniform memory architecture (NUMA). Many recent multicore systems are NUMA, Figure 1 shows an example NUMA system with 2 processors and 8 cores. In

NUMA systems each processor has an on-chip memory controller (MC), thus a part of the main memory is directly connected to each processor. This layout offers high-bandwidth and low-latency accesses to the directly-connected part of the main memory. However, as each processor must be able to access the main memory of the other processor as well, processors are connected by a cross-chip interconnect (IC) and transferring data through the cross-chip interconnect adds overhead relative to transferring data on the on-chip memory controller, thus *remote main memory accesses* have larger latencies than *local main memory accesses* and *remote cache accesses* are more expensive than *local cache accesses* (Figure 1 shows the latencies associated with each of the four possible data sources on the Intel Nehalem, the values are based on [7]). For good memory system performance on NUMA systems it is crucial to co-locate computations with the data they access, an aspect not of importance on previous microprocessor generations; understanding the interaction between software and a NUMA memory system is necessary for providing good performance scalability for parallel programs on recent architectures.

This paper presents a study that characterizes the memory system performance of well-known PARSEC benchmark programs [2] on a recent generation of multicore-multiprocessor systems with a non-uniform memory architecture and aggressive prefetcher units. The paper has three main contributions. First, the paper presents a detailed analysis of the memory system performance of three PARSEC programs, `streamcluster`, `ferret`, and `dedup`. Based on the analysis we identify and quantify two factors, *program-level data sharing* and *irregular memory access patterns*, that limit the performance of multithreaded programs on modern NUMA-multicore systems. Second, the paper describes three simple source-level techniques to eliminate program-level data sharing and irregular memory access patterns: (1) *controlling the data distribution* of the program to make sure that memory regions are allocated at well-defined and distinct processors, (2) *controlling the computation distribution* of the program to guarantee that computations operate on distinct subsets of program data, and (3) *regularizing memory access patterns* so that the access patterns are easier to predict for processor prefetcher units. All techniques rely either on using existing OS functionality or on simple algorithmic changes to the program and result in performance improvements of up to 3.1X on recent NUMA-multicore systems. Third, as the proposed techniques effect many layers of a modern NUMA-multicore memory system, the last part of the paper reports detailed measurements of orthogonal experiments to quantify how these changes individually effect each distinct layer.

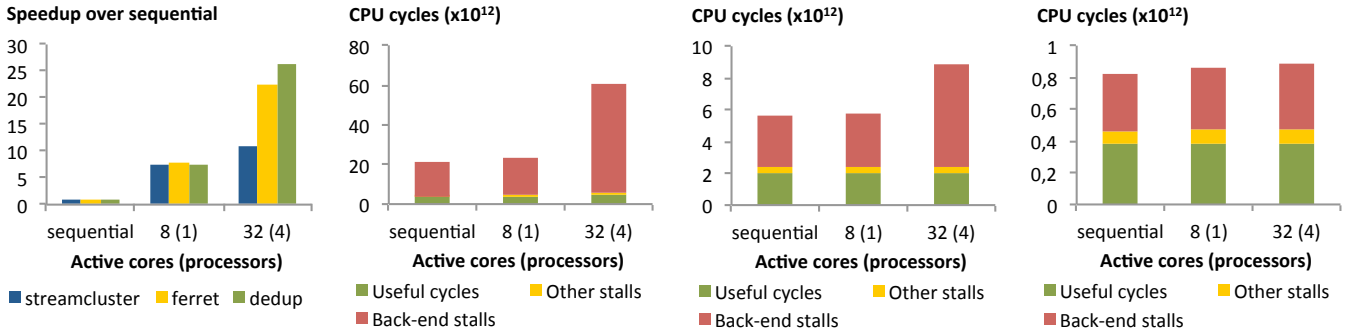
The PARSEC benchmarks are often used in a research setting to evaluate compiler/runtime optimizations [1]. By using the PARSEC programs out-of-the-box on recent NUMA-multicore systems, researchers might be misled to conclude that the microprocessor design, the compiler, or the runtime system are faulty when actually the application software contains performance-limiting factors to be blamed for unsatisfactory performance. The measurement methodology presented in this paper allows for pinpointing software-related problems detrimental to NUMA-multicore memory system performance, furthermore, we show that in many cases minor program-level adjustments suffice to fix these problems. There are scenarios when it is important to use an out-of-the-box version of an application for performance evaluation. But when, e.g., in a research setting, new features are to be evaluated, it is essential to identify limitations of an application's implementation that stand in the way of meaningful performance measurements. Publishing not only the results of the evaluation but also the software-level changes required to achieve them is necessary to fully understand a workload's performance characteristics.

## II. Motivation

To understand the performance of the PARSEC programs selected (`streamcluster`, `ferret`, and `dedup`), we execute these programs on a recent 4-processor 32-core machine based on the Intel Westmere microarchitecture. The memory system of the 4-processor Westmere machine is similar to that of the Nehalem system shown in Figure 1, however the Westmere processor allows additional cross-chip interconnects as there can be a point-to-point interconnect between any two of the four processors of the system (see Section III for more details).

Figure 2(a) shows the performance scaling of the three benchmark programs on the Westmere-based system. For each program we report performance in three cases. The first case is **sequential execution** when the non-parallelized version of the program is executed on a single core/processor. In the second case, **8-core (1 processor)**, the parallel version of the program is executed on 8 cores, but on a single processor so that there is no cross-chip traffic in the system. The third case, **32-core (4 processors)**, considers the scenario when all cores (and thus all processors) of the system are used to execute program threads. Figure 2(a) shows that the performance of all three programs scales well to 8 cores (1 processor), but the picture is different in the 32-core (4 processors) case: `streamcluster` shows bad performance scaling (11X speedup over single-threaded execution), `ferret` scales better (20X speedup), and `dedup` scales well (26X speedup).

There are many possibly reasons for the (non-)scalability of parallel programs (e.g., serialization due to extensive synchronization, load unbalance), and there exist many techniques that target these problems. A crucial and interesting issue is, however, how inefficiencies related to the memory system effect performance scaling on NUMA-multicores. To get more insight we break down the total number of CPU cycles of each program into three categories: (1) **useful**




(a) Performance scaling. (b) Cycles: `streamcluster`. (c) Cycles: `ferret`. (d) Cycles: `dedup`.

Figure 2: Performance scaling and cycle breakdown.

**cycles** when the CPU makes progress, (2) **back-end stalls**, when execution is stalled because the processor's back-end cannot execute instructions due to the lack of hardware resources, and (3) **other stalls** (e.g., instruction starvation).

In Intel Westmere processors the processor back-end can stall due to many reasons (including the memory system), however, due to limitations of the Westmere's performance monitoring unit we cannot precisely measure the number of back-end stalls related to the memory system. Nevertheless, as we change only the way a program is mapped onto the architecture (but not the instruction stream generated by the programs), we suspect a change in the number of back-end stalls to be related to memory system inefficiencies.

Figure 2(b) shows that the sequential version of `streamcluster` spends already a large fraction of its cycles on waiting for the back-end. We measure the same amount of back-end stalls for the case when the parallel version of the program is executed on a single processor (the 8-core (1-processor) case). However, when the program is executed on multiple processors, there is almost a 3X increase in the number of back-end stalls. `ferret` has similar, yet not as pronounced problems (2X more back-end stall cycles in the multiprocessor configuration than in the single-processor configurations, as shown Figure 2(c)). The performance of `dedup` is not affected in the multiprocessor configuration, as its number of back-end stalls increases only slightly when executed on multiple processors (Figure 2(d)). In summary, as long as a program uses only one processor in a NUMA-multicore system, the performance of the parallel and sequential version of the program are similar, but if all processors are used, performance can unexpectedly degrade, most likely due to program using the memory system inefficiently. To determine the exact cause further investigation is needed, Sections III and IV present details of this investigation.

## III. Experimental setup

### A. Hardware

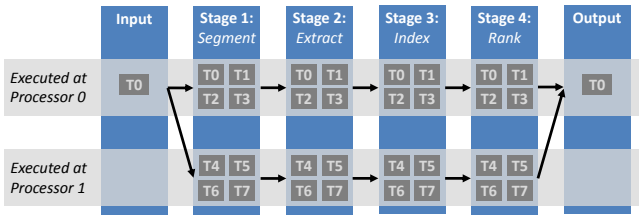
Two machines are used for performance evaluation: a 2-processor 8-core (Intel Nehalem microarchitecture) and a 4-processor 32-core (Intel Westmere microarchitecture) machine (see Table I for details about the systems). In both systems there is a point-to-point connection between every pair of processors, thus every processor is at a one-hop distance from any other processor of the system. We disable frequency scaling on both machines [4]. Processor prefetcher units are on if not stated otherwise.

| | Xeon E5520 | Xeon E7-4830 |
|---|---|---|
| Microarchitecture | Nehalem | Westmere |
| # of processors | 2 | 4 |
| Total # of cores | 8 | 32 |
| Clock frequency | 2.26 GHz | 2.13 GHz |
| Main memory | 2x6 GB DDR3 | 4x16 GB DDR3 |
| Cross-chip interconnect | 5.86 GTransfers/s | 6.4 GTransfers/s |
| Last-level cache | 2x8 MB | 4x24 MB |

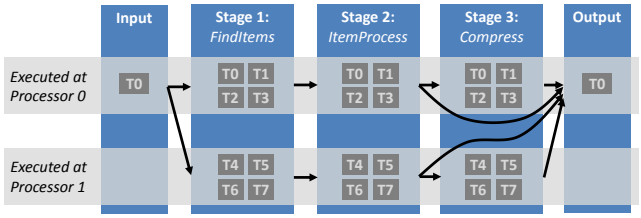
Table I: Hardware configuration.

### B. Benchmark programs

We consider three programs of the PARSEC benchmark suite [2]. (1) `streamcluster` is a data-parallel program that solves the on-line clustering problem for a set of input points. The program consists of a series of processing loops. These loops are parallelized with OpenMP directives, thus each worker thread is assigned a set of loop iterations. If not specified otherwise, `streamcluster` is configured with a number of worker threads equal to the number of cores of the machine it executes on (so that it makes optimal use of the computational resources offered by the architecture). (2) `ferret` is a pipeline-parallel program [11], [18], [15], [6], [8] that implements content-based similarity search of images: given a set of input images `ferret` searches in an image database for images similar to the input image. The program is structured as a set of 6 stages, as shown in Figure 3(a). Stages are interconnected by queues, data "flows" from the input stage through intermediary stages towards the output stage of the pipeline. Each pipeline stage of `ferret` is executed by a number of threads equal to the number of cores in the system; the input and the output stage is executed by a single thread each. Figure 3(a) shows the




(a) `ferret`.




(b) `dedup`.

Figure 3: Structure and configuration of pipeline programs.

runtime configuration of `ferret` on a 2-processor 8-core machine. (3) `dedup` is also a pipeline-parallel programs that consists of 5 stages. `dedup` compresses and deduplicates files given to it as input. The structure and runtime configuration of `dedup` (after cleanup) is shown in Figure 3(b)) and it is similar to that of `ferret`. Table II lists the benchmark programs together with the inputs considered. We have grown the size of the inputs relative to *native*, the largest input size available in the PARSEC suite. Most native-sized inputs fit into the last-level caches of the systems considered (e.g., the 32-core machine has 96 MB of total last-level cache), thus growing the size of the inputs allows us to exercise the memory system of the machines.

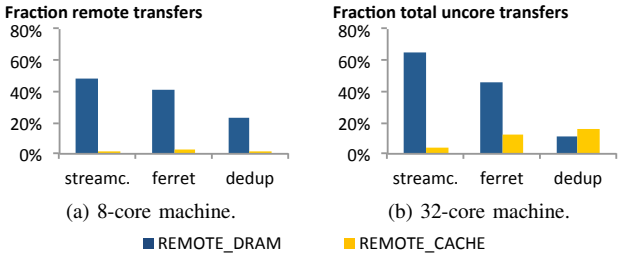
| Program | Input | Run time 8-core | Run time 32-core |
|---|---|---|---|
| `streamc.` | 10 M input points | 1232 s | 937 s |
| `ferret` | database (700 M images) and 3500 input images | 292 s | 133 s |
| `dedup` | 4.4 GB disk image | 46 s | 14 s |

Table II: Benchmark inputs and run times with default setup.

### C. Scheduling and memory allocation

In NUMA systems the placement of memory pages has a large impact on performance. Both machines that we use for evaluation run Linux, which, similar to other OSs, relies on the *first-touch page placement policy*. According to this policy each page is placed at the processor that first reads from/writes to the page after it has been allocated.

In NUMA systems not only the placement of memory pages at processors, but also the mapping of program threads to cores impacts program performance. In their default configuration OSs tend to change the thread-to-core mapping during the execution of programs. Such changes result in large performance variations. To reduce the negative effect




(a) 8-core machine. (b) 32-core machine.

Figure 4: Remote transfers as fraction of all uncore transfers.

of OS reschedules we use *affinity scheduling with identity mapping*. Affinity scheduling restricts the execution of each thread to a specific core. For `streamcluster` worker threads that execute a parallel loop are always mapped to cores so that T0 is executed at Core 0, T1 is executed at Core 1, etc. For `ferret` and `dedup` identity mapping is used for each pipeline stage individually (i.e., for each pipeline stage the worker threads executing that stage will be mapped to cores using identity mapping).

Identity mapping defines not only thread-to-core mappings, but implicitly also the thread-to-processor mapping (in NUMA systems, memory is distributed on a per-processor basis and not on a per-core basis). By using identity affinity there is an equal number of threads executing each parallel loop (in case of `streamcluster`) and each stage (in case of the pipeline programs) at all processors of the system. E.g., on the 2-processor 8-core system threads T0–T3 of each parallel loop/stage execute on Processor 0, and threads T4–T7 of each parallel loop/stage execute on Processor 1. Figure 3(a) and Figure 3(b) show how threads are mapped to processors for the `ferret` and `dedup` benchmark, respectively. Using identity mapping for mapping threads to processors/cores reduces measurement variation, as noted in [25].

## IV. Understanding Memory System Behavior

This section analyzes the memory system performance of the benchmarks (data locality and prefetcher effectiveness).

### A. Data locality

To understand how a NUMA-multicore memory system is used by the PARSEC programs we measure the read memory transfers generated by the programs on the uncore of the evaluation machines. The uncore of a processor includes the processor's last-level cache, its on-chip memory controller, and its interfaces to the cross-chip interconnect. Uncore transfers are local/remote main memory transfers and local/remote cache transfers[1]. Limitations of these processors' performance monitoring unit restrict the kind of information that can be obtained: only read transfers can be measured on Nehalem/Westmere processors.

[1]For space reasons we present only the case of a 2-processor 8-core machine; see Figure 1 for the four possible sources on such a system.

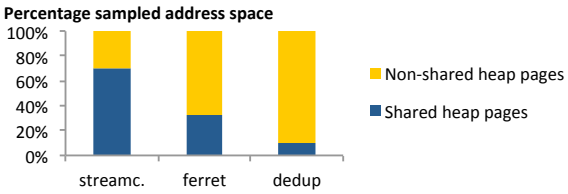



Figure 5: Data address characterization (8-core machine).

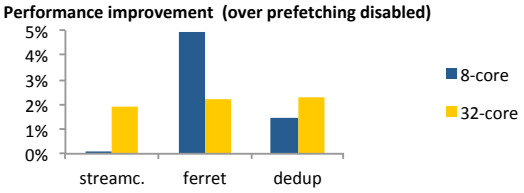



Figure 6: Performance gain w/ prefetching.

Figures 4(a) (4(b)) show the fraction of remote memory transfers relative to the total number of memory transfers generated by the programs on the 8-core (32-core) machine. On both machines a large fraction (11–67%) of the program's memory transfers are handled by remote main memory. Moreover, in some configurations, the programs show a large fraction of remote cache accesses, up to 16% of the total number of uncore transfers of the program. The measurements indicate that the PARSEC programs use an ill-suited data distribution (i.e., program data frequently resides in memory that is remote to the threads accessing it). Using affinity scheduling has the consequence that OS reschedules do *not* introduce remote memory transfers (as there are none). To understand the reasons for the ill-suited data distribution, we perform data address profiling on all three programs.

Data address profiling is a sampling-based approach that records the target data address (and thus the target memory page) of each sampled load operation [3], [10], [9], [5]. As data address profiling is sampling-based, it only approximates program behavior, nonetheless profiling data addresses still provides a good indication of where to concentrate optimization effort. Similar to the MemProf profiler [9], this profiler uses a hardware performance-counter based mechanism to gather address profiles. The mechanism used is specific to Intel processors, but other microarchitectures support data-address profiling as well (e.g., AMD processors implement Instruction-Based Sampling that offers functionalities similar to the Intel implementation of data address profiling). Because of limitations of the Linux performance monitoring system programs are profiled only on the 2-processor 8-core machine.

Based on their target data address, we group memory accesses into two categories: (1) **accesses to non-shared heap pages** (pages where accesses only by a single processor are recorded), and (2) **accesses to shared heap pages** (pages accessed by both processors of the NUMA system). Accesses to pages that hold the program's image, dynamically-linked shared objects and the program's stack are excluded from profiles.

Exclusively accessed heap pages are placed appropriately in processor memories by the first-touch page placement policy, as these pages are accessed only by a single core/processor. Moreover, data from private and exclusively accessed pages is present in only one cache of the system, thus these pages are likely not to be accessed from remote caches. However, for shared pages it is difficult to find an appropriate placement in a NUMA system: no matter at which processor of the system a shared page is placed, there will be accesses to these pages from a remote processor, as all processors access the shared data during the lifetime of the program. Likewise, the contents of shared pages is likely to be present in multiple caches of the system at the same time, thus reducing the capacity available to non-shared data and also leading to remote cache references.

Figure 5 shows the results of data address profiling for the PARSEC programs: a large fraction (10–70%) of their memory accesses are to problematic shared heap locations. Investigation of the address profiles and of the program source code reveals that data sharing is caused by data structures being shared between all threads of the program: As program threads execute at different processors, some threads are forced to access the shared data structure through the cross-chip interconnect. In the case of `streamcluster`, data sharing is caused by accesses to the data structure holding the coordinates of processed data points. In the case of `ferret`, data sharing is caused by lookups in the shared image database. In the case of `dedup`, most shared memory accesses are caused by accesses to a shared hash table that stores compressed chunks of the input file.

### B. Prefetcher behavior

To assess the effectiveness of the hardware prefetchers, we measured the performance of each benchmark in two configuration: with disabled processor prefetcher units and with the prefetcher units enabled. Figure 6 shows the performance improvement gained due to prefetching. In general, prefetchers improve performance, however the amount of performance improvement is small in most cases (5% at most), and in some cases prefetchers do not effect performance at all. The ineffectiveness of hardware prefetching can be explained by the irregular access patterns of the programs considered: `streamcluster` accesses the data points randomly, and both `ferret` and `dedup` store/look up data in hash tables at locations that look random to prefetchers that are good in recognizing strided data access patterns [24].

## V. Program transformations

The previous section has shown that two factors, program-level data sharing and irregular memory access patterns, cause the poor performance scalability of the three PARSEC programs on NUMA systems. These performance-limiting factors were inserted accidentally by the programmers who wrote these programs. The previous generation of multicore-multiprocessors had a uniform memory architecture (e.g., symmetric multiprocessors), thus non-uniform memory access times were not a problem for performance and are not considered in the source code of the PARSEC programs. The inclusion of aggressive prefetching into microprocessors is also relatively recent and it is rarely considered by programmers.

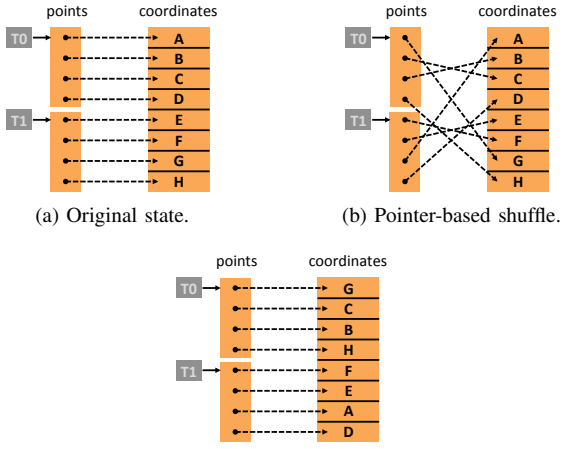
This section describes a set of simple source-level changes that allow the previously considered programs to better use the memory system of NUMA-multicores. The changes rely on a combination of using standard OS functionality to distribute memory pages across processors (Section V-A) and on simple algorithmic changes to the programs (Section V-B).

### A. Memory management in NUMA systems

To avoid remote main memory accesses the programmer must control at which processor data is allocated. Current operating systems provide support for per-processor memory allocation. E.g., Linux provides the `numa_alloc()` function that allows the programmer to allocate memory at a specific processor. Because of its high overhead, however, it is recommended to use this allocator only for the allocation of large memory regions. On Linux, once pages of a memory region are allocated, they can be migrated to a specific processor using the `move_pages()` system call. For the performance-critical memory regions of the programs considered we define a data distribution either by using `numa_alloc()` or by using memory migration. Defining the data distribution for these regions adds overhead, however, as we enforce the distribution only once, at program startup, the overhead is well compensated for by improved execution times. The overhead is included in the performance measurements.

### B. Algorithmic changes

*1) streamcluster:* The performance analysis described in Section IV reveals that most memory accesses of the program are to coordinates of data points processed by the program. The coordinates are stored in an array, `coordinates`. This array is accessed through a set of pointers stored in the `pointers` array. Figure 7(a) shows the two data structures assuming that the program is executed by two threads. Because of data parallelism each thread owns a distinct subset of pointers (and thus accesses a distinct set of points and coordinates). The situation with a higher number of threads is analogous to the two-thread case.



(a) Original state.

(b) Pointer-based shuffle.

(c) Copy-based shuffle.

Figure 7: `streamcluster`: Shuffles.

Investigation of the program source reveals that the bad memory system performance of `streamcluster` is caused by a data shuffling operation: during program execution the set of data points processed by the program are randomly reshuffled so that clusters are discovered with higher probability. Figure 7(a) shows the state of the `points` and `coordinates` array when the program starts, Figure 7(b) shows the state of the program data after the first shuffle. After shuffling each thread accesses the same subset of the `pointers` array as before the shuffle, however, the set of coordinates accessed is different. Shuffling is performed several times during the lifetime of the program.

The shuffling operation has negative effects on caching, prefetching and also data locality. As the set of data accessed by threads changes during program execution, the program does not have much data reuse. The presence of random pointers makes the access patterns of the program unpredictable and thus prefetching is not useful for this program. Lastly, due to the shuffles, each piece of coordinate data is possibly accessed by all threads of the program, thus coordinate data is shared between processors and it is impossible to distribute these data across the processors of the system. As a result, most accesses to the `coordinates` array are remote.

To enable memory system optimizations we change the way shuffle operations are performed. Instead of shuffling pointers so that they point to different data, we keep the pointers constant and move data instead. Figure 7(c) shows the data layout of `streamcluster` after a copy-based shuffle is performed on the original data shown in Figure 7(a). After the copy-based shuffle, the location of the coordinate data in memory accessed by each thread is the same as before the shuffling operations, however, the contents of the data locations has changed. As a result,

```
1  void shuffle() {
2    for (i = 0; i < NUM_POINTS; i++) {
3      j = random() % NUM_POINTS;
4      temp = points[i].coordinates;
5      points[i].coordinates = points[j].coordinates;
6      points[j].coordinates = temp;
7    }
8  }
```

(a) Shuffle (pointer-based implementation).

```
1  void shuffle() {
2    for (i = 0; i < NUM_POINTS; i++) {
3      j = random() % NUM_POINTS;
4      memcpy(temp, points[i].coordinates, BLOCK_SIZE);
5      memcpy(points[i].coordinates,
6             points[j].coordinates,
7             BLOCK_SIZE);
8      memcpy(points[j].coordinates, temp, BLOCK_SIZE);
9    }
10 }
```

(b) Shuffle (copy-based implementation).

Figure 8: `streamcluster`: Program transformation.

during the lifetime of the program each thread will access the same memory locations. Because data locations are not shared between threads they can be distributed appropriately between processors. Moreover, as each thread accesses its pointers sequentially, the access patterns of the program are more regular, thus prefetchers have a better chance to predict what data will be accessed next.

Figure 8(a) shows the code of the original, pointer-based shuffle, Figure 8(b) shows the code of the copy-based shuffling operation. Copy-based shuffling involves `memcpy` operations that is more costly than switching pointers, however shuffling is infrequent relative to data accesses, thus this change pays off (see the performance evaluation in Section VI).

*2) `ferret`:* The performance analysis described in Section IV indicates that `ferret`'s remote memory accesses are due to accesses to the shared image database. The image database is queried in Stage 3 (the indexing stage) of the program, Figure 9(a) shows the original pipelined implementation of this stage. Stage 3 (and other stages as well) receives the processor number $p$ as parameter (line 1). As a result, each stage instance is aware of the processor as where it is executing (the processor $p$ is specified by affinity scheduling with identity mapping). To keep lock contention low, two adjacent stages are usually interconnected by multiple queues, and only a subset of all threads executing the stage uses the same queue. Thus, each stage instance receives/puts data from/into queues local to that processor (lines 2, 4). Database queries are implemented by the `index()` function (line 3).

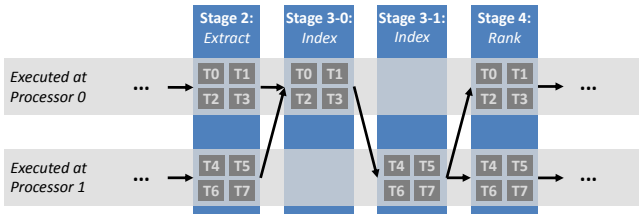
A way to reduce data sharing for the `ferret` benchmark is to partition the image database between the processors of the system, so that each processor holds a non-overlapping subset of the image database. Partitioning is performed at program startup when the image database is populated. Per-processor allocation (described in Section V-A) is used

```
1  void index_pipelined(Processor p) {
2    features = indexQueue[p].dequeue();
3    candidateList = index(features);
4    rankQueue[p].enqueue(candidateList);
5  }
```

(a) Stage 3 (original implementation).

```
1  void index_pipelined(Processor p) {
2    features = indexQueue[p].dequeue();
3    if (p == Processor.MIN) candidateList = 0;
4    candidateList += index(p, features);
5    if (p < Processor.MAX − 1) {
6      dstProcessor = p + 1;
7      indexQueue[dstProcessor].enqueue(candidateList);
8    } else {
9      rankQueue.enqueue(candidateList);
10   }
11 }
```

(b) Stage 3 (optimized implementation).




(c) Stages 2, 3, and 4 (optimized implementation).

Figure 9: `ferret`: Program transformation.

to ensure that each subset is allocated at the appropriate processor. The image database uses a hash-based index for lookup that is modified as well to include information about the processor at which each image is stored.

Database queries must also be changed to support data locality, Figure 9(b) shows the locality-aware implementation of Stage 3. The `index()` function (line 4) is modified to operate only on a subset $p$ of the image database ($p \in$ [0,`Processors.MAX`) where `Processors.MAX` is the total number of processors in the system). As a result, a stage instance executing at Processor $p$ performs database queries only on partition $p$ of the image database. Instances of Stage 3 are executed at every processor of the system (due to identity scheduling) thus all database partitions are eventually queried by the program. The final results of a query are composed from the results obtained by querying each partition of the complete image database.

To make sure that each query is performed on all database partitions, the queuing system is modified to dispatch queries to the appropriate processor. The partial result (stored in `candidateList`) of a query on partition $p$ of the image database is forwarded to an indexing stage executing at the next processor (i.e., the candidate list is forwarded from Processor $p$ to Processor $p + 1$) (lines 6–7 of Figure 9(b)). When all subsets of the image database have been queried, the candidate list contains the result of the query on the complete database and it is sent to the ranking stage (line 9).

All queries must start with the first partition of the image database. To ensure this, Stage 2 is modified to dispatch data to instances of Stage 3 executing at Processor 0. Similarly, to ensure load balance, the instances of Stage 3 querying the last partition of the image database dispatch data to all instances of Stage 4 (executing on all processors). As these changes are minimal they are not shown in the code example. Figure 9(b) shows a graphical representation of Stages 2, 3, and 4 of the optimized program as they are executed on a 2-processor system, this representation indicates the change of queuing in all stages.

The example shown in Figure 9(b) is for a 2-processor system. On this system the proposed optimization corresponds to creating two copies of Stage 3, where one copy of this stage, Stage 3-0, executes at Processor 0, and the other copy, Stage 3-1, executes at Processor 1. This solution scales with the number of processors (i.e., on $n$ processors $n$ copies of Stage 3 are created).

*3) dedup:* Figure 10(a) shows the pipelined implementation of Stage 1 and Stage 2 of `dedup`. The performance analysis described in Section IV indicates that accesses to the shared hash table (line 12–13) are the source of data sharing for `dedup`. Not just data items, but also the key of every data item (initialized in line 11) is added to the hash table. As these keys are used frequently for hash table lookups, the processor that holds each key also plays an important role for data locality.

We change `dedup` to reduce sharing of hash table elements and also of the hash table structure itself, the changes are illustrated in Figure 10(b). The set of all hash keys are divided into $p$ non-overlapping subsets, where $p$ is the number of processors in the system. Each subset is (conceptually) mapped to a different processor. As this mapping closely depends on the structure hash table, we add a new method to the hash table, `processorForKey()` that determines for each key the processor associated with that key (line 7).

Enforcing the mapping of keys to processors requires data elements to be dispatched to the appropriate processor. We modify Stage 1 of `dedup` to achieve this. The original version of Stage 1 dispatches each data item to the same processor as where it was dequeued from (lines 3 and 6 of Figure 10(a)). In the optimized version, however, each data item is dispatched to the processor its key is associated with (lines 7–8 of Figure 10(b)). It is sufficient to change Stage 1 to enforce the mapping of keys to processors, because by default Stage 2 and Stage 3 enqueue each data elements at the same processor as where they dequeued it from, thus as soon as a data element is enqueued at the appropriate processor by Stage 1, it is processed at the same processor by all subsequent stages.

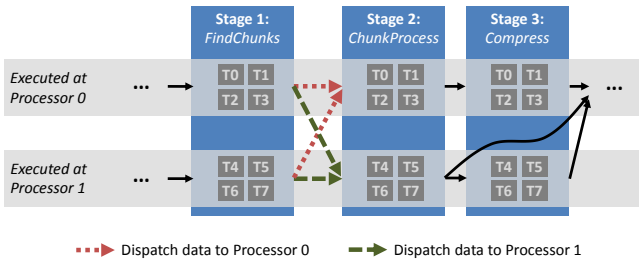
Using locality-aware dispatching of data elements guarantees data locality for structures related to the hash table but not for keys. As keys are computed before the dispatch

```
1  // Stage 1: Divide data chunk into items.
2  void findItems_pipelined(Processor p) {
3    fileChunk = inputQueue[p].dequeue();
4    items = findItems(fileChunk);
5    for (item : items)
6      itemProcessQueue[p].enqueue(item);
7  }
8  // Stage 2: Process an item.
9  void itemProcess_pipelined(Processor p) {
10   item = itemProcessQueue[p].dequeue();
11   item.key = getSHA1Signature(item);
12   if ((entry = hashTable.get(key)) != null) {
13     entry.count++;
14     outputQueue[p].put(item.key);
15   } else {
16     compressQueue[p].put(item);
17   }
18 }
```

(a) Stages 1 and 2 (original implementation).

```
1  // Stage 1: Divide data chunk into items.
2  void findItems_pipelined(Processor p) {
3    fileChunk = inputQueue[p].dequeue();
4    items = findItems(fileChunk);
5    for (item : items) {
6      item.key = getSHA1Signature(item);
7      dstProcessor = hashTable.processorForKey(key);
8      itemProcessQueue[dstProcessor].enqueue(key);
9    }
10 }
11 // Stage 2: Process an item.
12 void itemProcess_pipelined(Processor p) {
13   item = itemProcessQueue[p].dequeue();
14   item.key = clone(item.key);
15   if ((entry = hashTable.get(item.key)) != null) {
16     entry.count++;
17     outputQueue[p].put(item.key);
18   } else {
19     compressQueue[p].put(item);
20   }
21 }
```

(b) Stages 1 and 2 (optimized implementation).




(c) Stages 1, 2, and 3 (optimized implementation).

Figure 10: `dedup`: Program transformation.

to subsequent stages can happen (line 6 of Figure 10(b)), the keys are not necessarily allocated at the processor that they are conceptually mapped onto. To enforce the mapping of keys to processor, a clone of each key is created after dispatch (in Stage 2, cloning shown in line 14 of Figure 10(b)). As Stage 1 already enforces the previously described key-to-processor mapping, the clones created in Stage 2 are guaranteed to reside on the appropriate processor due to the first-touch page placement policy. Figure 10(c) shows a graphical representation of the locality-aware implementation of stages 1, 2, and 3 of `dedup`, including queuing.

## VI. Performance evaluation

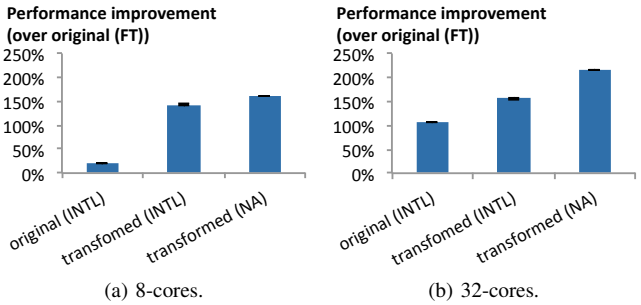
This section evaluates the performance of the program-level transformations proposed in Section V. As the transformations effect both data locality and prefetcher performance, we divide the evaluation into two parts: first we discuss the combined effect of the program transformations (data locality and prefetcher performance), then we quantify the effect of the transformations on prefetcher performance individually.

### A. Cumulative effect of program transformations

This section evaluates the effect of the transformations on both data locality and prefetcher performance cumulatively, thus the prefetcher units of the processors are turned on for the measurements presented in this section. We evaluate the performance of the program transformations in different execution scenarios. An execution scenario defines three aspects: (1) which version of the program is used (original or transformed), (2) the way a program allocates memory (i.e., the page placement policy used), and (3) how a program's threads are scheduled. We use affinity scheduling with identity mapping in all execution scenarios, therefore execution scenarios differ in the first two aspects.

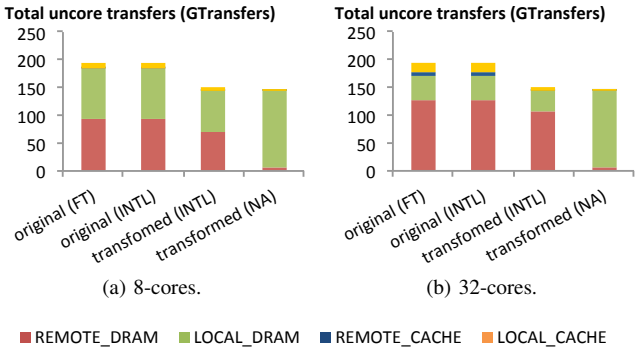
We consider four execution scenarios. **(1) original (first-touch)** This execution scenario uses the original, out-of-the-box version of the programs. The memory allocation policy used is first-touch (the default on many modern OSs). On the figures this scenario appears as *original (FT)*. **(2) original (interleaved)** This execution scenario also uses the original version of the programs, but memory regions are allocated using the interleaved page placement policy. The *interleaved page placement policy* distributes pages of shared memory regions across the processors of the system in a round-robin fashion. Interleaved page allocation balances the load between the memory controllers of a NUMA system and thus mitigates the performance degradation caused shared data regions being allocated at a single processor, as shown in [9], [5]. For this scenario we use the shorthand *original (INTL)* on the figures. **(3) transformed (interleaved)** This scenario evaluates the proposed program transformations with the interleaved page placement policy. As per-processor memory allocation is not used in this scenario, performance differences are only caused by the combined effect of improved caching and prefetcher performance. On the figures this scenario appears as *transformed (INTL)*. **(4) transformed (NUMA-alloc)** This scenario evaluates the program transformations with per-processor memory allocation enabled. This scenario has the additional benefit of local main memory accesses relative to the previous, transformed (interleaved), execution scenario. On the figures this scenario appears as *transformed (NA)*.

We compare the performance of the three benchmarks, `streamcluster`, `ferret` and `dedup` in all execution scenarios on both machines described in Section III (2-processor 8-core and 4-processor 32-core NUMA). Any two adjacent execution scenarios differ in only one parameter, thus comparing any two adjacent scenarios quantifies the exact cause of performance improvement. Besides reporting performance we quantify the effect of the transformations on the memory system by providing a breakdown of total uncore memory transfers for all configurations of the benchmark programs. (A processor's uncore includes the processor's last-level cache, its on-chip memory controller, and its interfaces to the cross-chip interconnect, thus on the uncore level we can distinguish between local/remote cache accesses and local/remote main memory accesses.) In some cases the total number of uncore transfers differs when the same program binary is executed on two different machines but in the same execution scenario. Although the two machine used for evaluation have the same microarchitecture, the sizes of the last-level caches differ between the two machines that explains the difference.




(a) 8-cores. (b) 32-cores.

Figure 11: `streamcluster`: Performance comparison.




(a) 8-cores. (b) 32-cores.

Figure 12: `streamcluster`: Uncore memory transfers.

*streamcluster:* Figures 11(a) and 11(b) show the performance improvement of three scenarios, original (interleaved), transformed (interleaved), and transformed (NUMA-alloc), over the default setup (original (first-touch)). Figures 12(a) and 12(b) show the breakdown of all uncore transfers for all the scenarios considered.

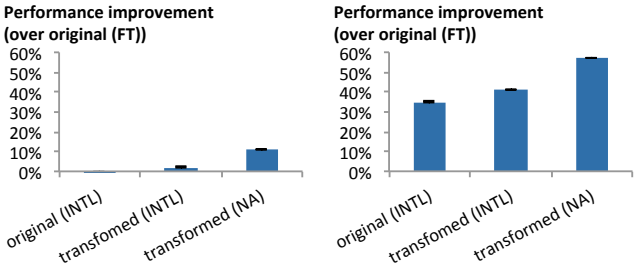
By default the complete coordinates of all data points accessed by `streamcluster` are allocated at a single processor. As all threads access these data, the memory

controller and cross-chip interconnect of this processor are overloaded. Using the interleaved page placement policy instead of the first-touch page placement policy reduces neither the fraction of remote cache references nor the fraction of remote main memory accesses relative to the first-touch page placement policy. However, as the interleaved policy distributes the pages of the image database across the processors of the system in a round-robin fashion, the load on the memory system is balanced between all memory controllers/cross-chip interconnects of the system. Using interleaved page placement for `streamcluster` has been previously suggested by Lachaize et al. in [9] and it results in a performance improvement of 21% (106%) relative to the default setup on the 8-core (32-core) machine. In light of these measurements one could question if the interleaved page placement policy [9] is appropriate and should be used in the future.

The next scenario considered is program transformations with interleaved page placement (transformed (INTL)). The total number of transfers executed by the program decreases relative to the previous scenario. This result indicates better cache usage. In addition, prefetchers are also able to better predict the memory accesses of the program (see Section VI-B). The combined result of these two effects result in an additional performance improvement of 121% (50%) on the 8-core (32-core) machine. If the data distribution of the program is also set appropriately (i.e., by using per-processor memory allocation), we record an almost complete elimination of remote main memory accesses. This setup is reported in the transformed (NA) execution scenario. For the transformed (NA) scenario we observe a performance improvement of 18% (58%) on the 8-core (32-core) machine, additionally to the program transformations (interleaved) scenario. In the end, the algorithmic changes coupled with per-processor memory allocation improve performance by 2.6X (3.14X) over the out-of-the-box version of the `streamcluster` benchmark on the 8-core (32-core) machine.

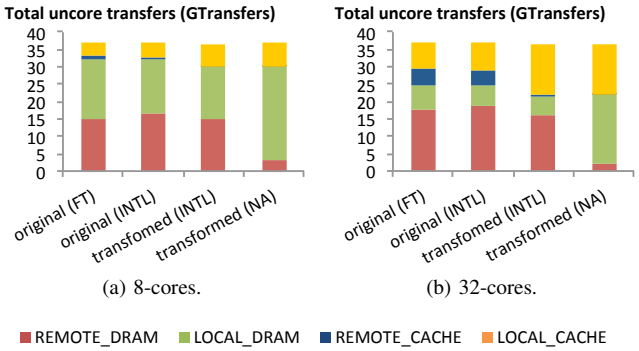
*ferret:* Figures 13(a) and 13(b) compare the performance of `ferret` in the four execution scenarios considered, Figures 14(a) and 14(b) present the breakdown of uncore transfers corresponding to each scenario.

With the default setup a large fraction of `ferret`'s memory accesses hit in the remote cache or are served by remote main memory, thus `ferret` experiences large performance penalties due to increased memory access latencies. By default, the complete image database is allocated at a single processor. As all threads use the database, the memory controller and cross-chip interconnect of this processor are overloaded. The interleaved policy balances the load between all memory controllers/cross-chip interconnects of the system, thus interleaved allocation results in a performance improvement of 35% relative to the default setup on the 32-core machine (but no improvement on the 8-core machine).




(a) 8-cores. (b) 32-cores.

Figure 13: `ferret`: Performance comparison.



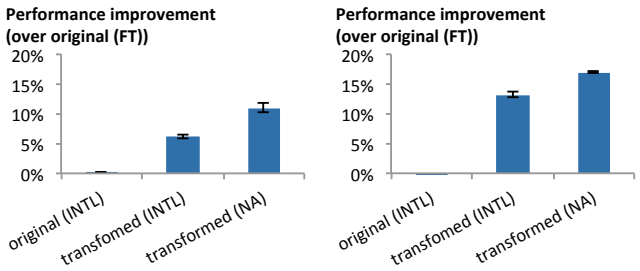

(a) 8-cores. (b) 32-cores.

Figure 14: `ferret`: Uncore memory transfers.

The next execution scenario considered is the transformed version of the program used with interleaved memory allocation. With this execution scenario each processor accesses only a subset of the complete image database thus costly remote cache accesses are almost completely eliminated. If data is shared between processors, each processor has a copy of the shared data in its cache. If data sharing is eliminated, each piece of data is present only in the cache of a single processor, thus there is more cache capacity available. Measurements show that cache hit rate of the program increases due to the program transformations; this results in an additional 2% (6%) performance improvement relative to the original (interleaved) execution scenario. Distributing pages correctly at the processors of the system (the transformed (NA) execution scenario) reduces the fraction of remote main memory references of the program and thus further improves performance by 9% (16%) on the 8-core (32-core) machine. In summary, due to all program-level changes the performance of `ferret` improves 11% (57%) on the 8-core (32-core) machine.[2]

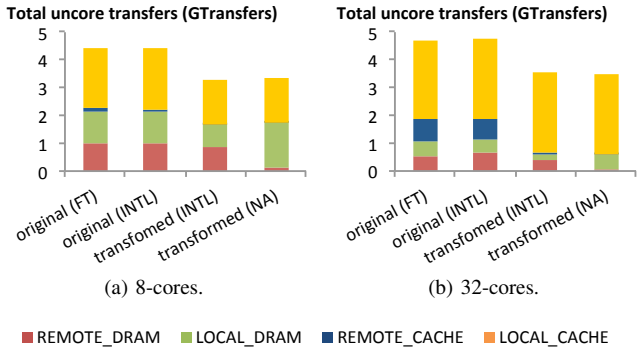
*dedup:* Figures 15(a) and 15(b) compare the performance of the four previously discussed execution scenarios for the `dedup` benchmark. Figures 16(a) and 16(b) show

[2]In the case of `ferret` another optimization opportunity is to replicate the shared image database at each processor. This optimization reduces the fraction of remote memory references, but also reduce the effectiveness of caching because each processor's last-level cache is used for caching a different replica. The results of replication are inferior to transformed (NA), which gets the benefits of both caching and few remote memory accesses.




(a) 8-cores. (b) 32-cores.

Figure 15: `dedup`: Performance comparison.




(a) 8-cores. (b) 32-cores.

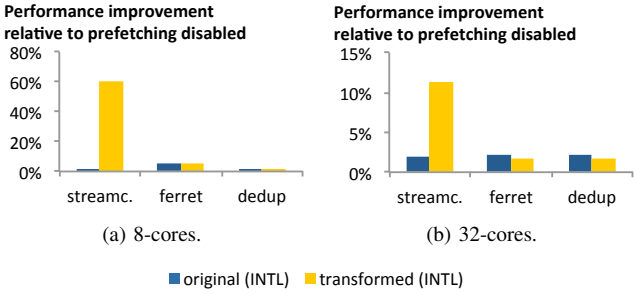
Figure 16: `dedup`: Uncore memory transfers.

the uncore traffic breakdown for each configuration.

The shared hash table is constructed during program execution, and all threads add new file chunks to the hash table as deduplication progresses. As a result, with the first-touch allocation policy the hash table is spread across the processors of the system. Therefore, all memory controllers/cross-chip interconnects of a system are used to access these data structures, thus none of the interfaces to memory are overloaded. As a result, using the interleaved page placement policy does not significantly change performance over using the first-touch policy. However, the transformed version of the program uses locality-aware dispatching of data to processors and thus each processor accesses only a subset of the globally shared hash table. As a result, the fraction of remote cache accesses and also the amount of uncore transfers generated decreases on both machines. Due to better caching, performance increases by 6% (13%) on the 8-core (32-core) machine. The last execution scenario considered is transformed with NUMA-aware allocation. Due to a reduction of the fraction of remote main memory accesses of the program, performance increases by an additional 5% (4%) on the 8-core (32-core) machine. In summary, the performance of `dedup` improves 11% (17%) on the 8-core (32-core) machine over the default.

### B. Prefetcher performance

The previous section evaluated in detail the performance benefits of the proposed program transformations, however,




(a) 8-cores. (b) 32-cores.

Figure 17: Prefetcher performance.

it did not clearly identify the benefits due to prefetching effects, because the transition from the original (INTL) execution scenario to the transformed (INTL) shows improvements due to both caching- and prefetching effects.

To quantify how much performance benefit there is due to the more regular access patterns caused by the source-level transformations, we perform a simple experiment: we execute each benchmark with the processor prefetcher units disabled and then with prefetching enabled. At the end we compare the performance of the two cases. The results are shown in Figure 17(a) (17(b)) for the 8-core (32-core) machine.

For the original version of the program (executed in the original (INTL) scenario) there is a moderate performance improvement due to prefetching (at most 5%). The transformed version of the program (executed in the transformed (INTL) scenario), however, shows performance improvements of 60% (11%) due to prefetching for the `streamcluster` benchmark on the 8-core (32-core) machine. For the other two benchmarks the benefit of prefetching is roughly the same as with the original version of the program. As both programs use a hash table-based data structure intensively, they both have irregular memory access patterns. The hash table is key to the algorithms implemented by the programs, similarity search based on locality-sensitive hashing in case of `ferret` and data deduplication in case of `dedup`. Replacing hash tables with a different data structure would change the essence of these programs and requires more significant changes than those presented by this paper.

## VII. Related work

There exist many automatic approaches to improve the data locality of multithreaded programs. OSs are in a good position to increase data locality by migrating data close to the threads using them [3] and by replicating data across processors [23], [5]. These approaches have large performance benefits, but they face limitations in case of programs with data sharing. Tang et al. [20] show that scheduling threads close to the data they access can be beneficial for data locality, but in some cases cache contention counteracts the benefits of data locality.

Profilers are also in a good position to optimize data placement multithreaded programs, as shown by Marathe et al. [10]. Page placement can also be performed at runtime based dynamic data access profiles [22], [13], [12]. Although these approaches are beneficial for performance, they work well only for programs that have little data shared between threads. Su et al. [17] show that dynamic data access profiles can be used to guide thread placement as well to improve data locality. Tam et al. [19] cluster the threads of a multithreaded program based on the amount of data they share. Threads of the same cluster are then scheduled onto cores connected to the same last-level cache to reduce the number of remote cache accesses. The programs considered by Tam et al. exhibit non-uniform data sharing (i.e., each thread shares data with only a subset of the threads of the program), uniform data sharing can hinder thread clustering, as noted by Thekkath et al. [21].

All previously mentioned are fully automatic, that is, the runtime system, the OS, or the compiler is able to perform the optimizations without programmer intervention. Unfortunately, however, data locality optimizations cannot be automated in many cases because they require high-level semantic information about the program. Previous work has shown that the memory system performance (and thus the run time) of many multithreaded programs can be improved on the source-code level by using compiler directives available in today's commercially available state-of-the-art compilers [16], by making simple algorithmic changes to the program [25], or by using a combination of the two [9]. Although all previously mentioned papers use recent multicore machines for evaluation, none of them considers all aspects related to cross-processor accesses on NUMAs: [16] and [25] do not consider NUMA at all; [9] does not take cache performance into account and in many cases it recommends disabling NUMA (i.e., using page-level interleaving) for problematic memory regions. Furthermore, none of the previously mentioned papers considers the behavior of the processor's prefetch units, an important aspect for the performance of today's processors [24].

## VIII. Conclusions

Program-level data sharing and irregular memory access patterns are two factors with a large influence on the performance of NUMA multicore systems. Existing benchmark programs, originally developed for earlier generations of symmetric multiprocessors, show sharing and access patterns that inhibit performance on modern platforms with advanced architecture features such as prefetch units. These factors must be taken into account when evaluating performance.

Our experience with programs from the PARSEC suite shows that programs can be adjusted with minimal effort to fit modern memory system architectures. The transformations result in better data locality (fewer accesses to remote data), better use of the caches, and increased effectiveness of the prefetch unit. Simple source-level changes result in high performance improvements (up to 3.1X) for three benchmark programs on recent NUMA-multicores. The paper reports detailed measurements of orthogonal experiments to quantify how these changes individually effect each distinct layer of a modern NUMA-multicore memory system. As we expect the performance gap between local and remote accesses to widen (with an increased number of processors and larger caches), it is important to use appropriate benchmarks for performance evaluation on future multicore-multiprocessors.

## Acknowledgements

We thank Albert Noll and the anonymous referees for their helpful comments.

## References

[1] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT '08*.
[3] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for NUMA-aware contention management on multicore processors," in *USENIX ATC '11*.
[4] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the Intel Core i7 Turbo Boost feature," in *IISWC '09*.
[5] M. Dashti, A. Fedorova, J. Funston, F. GAud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placmeent on numa systems," in *ASPLOS '13*.
[6] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS '06*.
[7] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *MICRO '09*.
[8] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *PACT '09*.
[9] R. Lachaize, B. Lepers, and V. Quma, "MemProf: a memory profiler for NUMA multicore systems," in *USENIX ATC'12*.
[10] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-directed page placement for ccnuma via hardware-generated memory traces," *J. Parallel Distrib. Comput.*, vol. 70, no. 12, pp. 1204–1219, Dec 2010.
[11] A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval, "Analytical modeling of pipeline parallelism," in *PACT '09*.
[12] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "A case for user-level dynamic page migration," in *ICS '00*.
[13] T. Ogasawara, "NUMA-aware memory manager with dominant-thread-based copying GC," in *OOPSLA '09*.
[14] M. Roth, M. Best, C. Mustard, and A. Fedorova, "Deconstructing the overhead in parallel applications," in *IISWC'12*.
[15] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *PACT '11*.
[16] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *ISCA '12*.
[17] C. Su, D. Li, D. S. Nikolopoulos, M. Grove, K. Cameron, and B. R. de Supinski, "Critical path-based thread placement for NUMA systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 106–112, Oct 2012.
[18] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *PACT '10*.
[19] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *EuroSys '07*.
[20] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing googles warehouse scale computers: The NUMA experience," in *HPCA '13*.
[21] R. Thekkath and S. J. Eggers, "Impact of sharing-based thread placement on multithreaded architectures," in *ISCA '94*.
[22] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *J. Parallel Distrib. Comput.*, vol. 68, no. 9, pp. 1186–1200, Sep 2008.
[23] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," in *ASPLOS '96*.
[24] C.-J. Wu and M. Martonosi, "Characterization and dynamic mitigation of intra-application cache interference," in *ISPASS '11*.
[25] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in *PPoPP '10*.