# ULCC: A User-Level Facility for Optimizing Shared Cache Performance on Multicores

Xiaoning Ding *
The Ohio State University
dingxn@cse.ohio-state.edu

Kaibo Wang
The Ohio State University
wangka@cse.ohio-state.edu

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

## Abstract

Scientific applications face serious performance challenges on multicore processors, one of which is caused by access contention in last level shared caches from multiple running threads. The contention increases the number of long latency memory accesses, and consequently increases application execution times. Optimizing shared cache performance is critical to significantly reduce execution times of multi-threaded programs on multicores. However, there are two unique problems to be solved before implementing cache optimization techniques on multicores at the user level. First, available cache space for each running thread in a last level cache is difficult to predict due to access contention in the shared space, which makes cache conscious algorithms for single cores ineffective on multicores. Second, at the user level, programmers are not able to allocate cache space at will to running threads in the shared cache, thus data sets with strong locality may not be allocated with sufficient cache space, and cache pollution can easily happen.

To address these two critical issues, we have designed ULCC (User Level Cache Control), a software runtime library that enables programmers to explicitly manage and optimize last level cache usage by allocating proper cache space for different data sets of different threads. We have implemented ULCC at the user level based on a page-coloring technique for last level cache usage management. By means of multiple case studies on an Intel multicore processor, we show that with ULCC, scientific applications can achieve significant performance improvements by fully exploiting the benefit of cache optimization algorithms and by partitioning the cache space accordingly to protect frequently reused data sets and to avoid cache pollution. Our experiments with various applications show that ULCC can significantly improve application performance by nearly 40%.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

***General Terms*** Algorithms, Design, Performance

***Keywords*** Multicore, Cache, Scientific Computing

## 1. Introduction

Multicore processors have been widely used in all kinds of computing platforms from laptops to large supercomputers. According to the recently announced Top 500 supercomputer list, by June 2010, 85% of these supercomputers have been equipped with quad-core processors and 5% use processors with six or more cores [27]. However, application programming is facing a new performance challenge on multicore processors caused by the bottleneck of the memory system (e.g. [19, 33]). On a multicore processor, the last level cache space and the bandwidth to access memory are usually shared and contended among multiple computing cores. The contention for the shared cache increases the amount of accesses to off-chip main memory, and the contention for memory bandwidth increases the queuing delay of memory accesses. The accumulated contention in the cache and the memory bus significantly delays the execution time due to inefficient management of the shared cache at runtime. Optimizing the performance of shared last level caches can reduce both slow memory accesses and memory bandwidth demand, which has become a critical technique to address the performance issue in multicore processors.

Cache optimization at the user level has been one of the most effective methods to improve execution performance for scientific applications on the platforms with single-core processors. A large number of research projects have been carried out to restructure algorithms and programming with cache optimization (e.g. [5, 8, 12, 20, 28–31]). However, cache optimization in multicore processors faces two new challenges due to architectural changes in the memory hierarchy.

One important factor for cache conscious programming is the available cache size for given data sets in order to fully utilize the cache with minimum misses [33]. However, on multicores, due to access dynamics to shared caches, the available cache space size for each thread can be hardly predicted, particularly when an application is programmed in a MPMD model (multiple programs co-run on the same set of processors). Let's take a blocking algorithm for linear systems as an example. In such an algorithm, the block size is an important factor affecting performance, and an unsuitable block size causes extra cache misses, leading to poor execution performance. An optimal block size is usually a function of the available cache space size for blocking. On a single-core processor the last level cache is not shared, and the available space for blocking is determined by the cache size. However, on a multicore platform, a last level cache is shared among multiple threads co-running on multiple cores. How much cache space a thread can occupy is determined by dynamic access patterns of the thread and other threads sharing the cache with it. Thus, it is difficult for a programmer to determine the available cache space for each thread to make effective blocking actions. In practice, a sub-optimal block size may be selected, causing mediocre or even poor performance.

Another major source of poor performance in a multicore processor is last level cache pollution, which is a more serious problem than that on a single-core processor because multiple tasks are affected. Cache pollution is incurred when a thread accesses a sizable

* Currently working at Intel Labs Pittsburgh.



*PPoPP'11*, February 12–16, 2011, San Antonio, Texas, USA.

data set with weak locality (i.e. data with infrequent reuses or without reuses), and consequently replaces data sets with strong locality (i.e. data with frequent reuses in the cache). For threads running on single-core processors, because of the private cache architecture, a thread can only pollute the cache on one core (processor) and cannot affect threads running on other cores (processors). However, on a multicore processor, because of the shared cache architecture, a thread can pollute the whole shared cache and affect all the other threads sharing the cache. Furthermore, on a multicore processor, multiple threads may access weak locality data sets simultaneously and evict strong locality data sets very quickly.

Due to the above serious concerns, it is highly desirable for programmers to distinguish weak locality data sets from strong locality data sets and explicitly specify different space allocation priorities to them in shared caches on multicore processors. In other words, a strong locality data set should be protected by allocating it with sufficient space, while a weak locality data set must be carefully watched by giving it limited space. Unfortunately, programmers lack necessary system support to make effective allocation actions even though the programmers are very knowledgeable about the locality strength of each data set.

To address these two critical issues, we present ULCC (User Level Cache Control), a software runtime library that enables programmers to explicitly manage space sharing and contention in last level caches by making cache allocation decisions based on data locality strengths. With the functions provided by ULCC, programmers can hand-tune their programs to optimize the performance of last level caches on multicores. Unlike database applications or server applications, whose access patterns are dynamic and sometimes determined by the distribution of their data or requests, most scientific applications have regular and consistent access patterns. Thus, scientific application programmers can determine the sizes and locality strengths of data sets based on their algorithms in the programming stage. With the locality information and our effective support from ULCC, programmers can make effective decisions and enforce a necessary cache space allocation for their programs that can facilitate cache optimization in the programming stage and ensure that strong locality data stay in the cache during executions.

We make three major contributions in this paper. First, we have carefully designed ULCC as a runtime library to enable user level cache controls for application programming. We provide a set of functions in ULCC to support different programming models, such as MPI, OpenMP, and pthread, to tightly couple our ULCC implementation with commonly used programming interfaces. With these functions, ULCC allows programmers to manage cache space allocation flexibly and effectively while it hiding most complexity of the cache structure on multicore architecture and ULCC implementation details. Thus, programmers can focus on analyzing their algorithms and planning optimal cache space allocation; while ULCC can focus on helping users making full utilization of cache space with least overhead. Second, we have implemented a prototype of ULCC at the user level based on operating system support. Though ULCC relies on the page coloring technique [15], it does not require OS kernel modifications. This makes ULCC highly portable. Finally, we have tested ULCC with extensive experiments as to its effectiveness in improving the performance of scientific programs. We have also evaluated the overhead of ULCC. Our experiments show that ULCC can effectively and significantly improve execution performance with negligible overhead.

The remainder of the paper is organized as follows. In Section 2, we introduce the motivation and the background information of ULCC. Then in Section 3 we present the overall structure of ULCC, the design of its key components, and the implementation based on the Linux system. We present our experience with ULCC in Section 4. Finally, we discuss related works in Section 5, and conclude the paper in Section 6.

## 2. Motivation and Background

In this section, with a motivating example, we illustrate the challenges a programmer may encounter in cache optimization. We show how ULCC works to help the programmer address the challenges, and explain the underlying techniques ULCC relies on to achieve this goal.

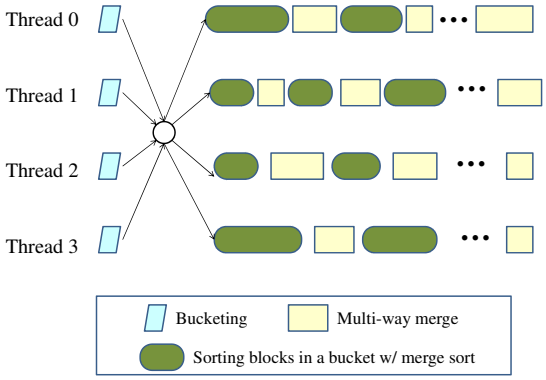


**Figure 1.** An application sorting a large array with multiple threads

The example program sorts elements in a large array with four threads in parallel on a quad-core processor. The program first rearranges the elements into a number of buckets according to the values of their keys in a way similar to bucket sort. Then it sorts the elements in every bucket with merge sort. To improve cache efficiency, optimizations including blocking and multi-way merging are applied as suggested in [13]. For each bucket, a thread first partitions the elements into blocks. Then it sorts the blocks one by one with merge sort. When a thread sorts a block, it uses a sorting buffer to store the intermediate results of merge sort. The buffer has the same size as the block size, and is reused for sorting different blocks. The block size should be adjusted to guarantee that sorting each block does not incur extra memory accesses after the block has been loaded into the last level cache. After all the blocks have been sorted, the thread merges the sorted blocks in one pass with a multi-way merging by constructing a full binary tree structure. Therefore, after the buckets are ready, each thread repeatedly selects an unsorted bucket, sorts the blocks in it, and merges the sorted blocks, as illustrated in Figure 1.

When the program runs on a quad-core X5355 processor in which there are two pairs of cores and cores in each pair share an L2 cache, the interference caused by cache contention and cache pollution can significantly slowdown its execution. Cache pollution happens when one thread is sorting blocks and the other thread sharing the same L2 cache with it is merging sorted blocks. Most of the data accessed by merging, including the sorted blocks and the buffer saving the final results, will not be reused. Accessing them means loading them into the L2 cache and evicting the to-be-reused data, e.g. the block being sorted and the sorting buffer.

Cache space contention happens when both threads sharing the same L2 cache work on sorting blocks. For the threads sharing the same L2 cache, if the aggregated size of their sorting buffers and the blocks they are sorting exceeds the L2 cache size, severe cache contention will occur. To quickly sort the elements in each block, the total size of the blocks being sorted and sorting buffers should fit into the last level cache. However, cache contention still happens when a thread finishes sorting a block and starts to work on another

block. Loading the new block into the L2 cache evicts the to-be-reused data in sorting buffers and the block being sorted by the other thread. This causes further performance degradation, as we will show in Section 4.

With the support from ULCC, the program can separate the cache space used by different threads and reserve separate cache space slots in the last level cache for each block being sorted and the sorting buffer for each thread, in order to avoid cache pollution and cache contention. Thus, each thread can carry out merge sort block by block without suffering interference from other threads or from its own block switching. With this method, the performance of the program can be improved by over 20%.

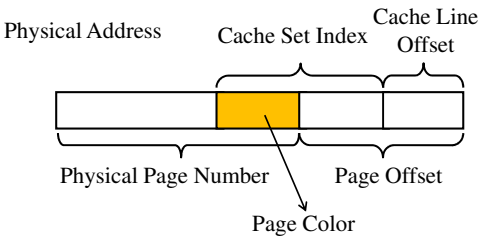


**Figure 2.** Physical address in the page coloring technique

ULCC enforces a user demanded cache allocation based on the page coloring technique. The page coloring technique was proposed for operating systems to reduce cache misses through careful mapping between virtual pages and physical pages for single-core [9] and multicore processors [15]. The logic of the page coloring technique is shown in Figure 2. Memory management in operating systems uses the most significant bits of a physical address as the physical page number. When the address is used in a cache lookup operation, some bits in the middle of the address (*cache set index* in the figure) are used to determine the cache set to look up. There are several common bits between the cache set index and the physical page number. These bits are referred to as page color. The page coloring technique assigns a page color to every physical page and cache set. Thus, it divides a cache into multiple non-overlapping bins (denoted as cache colors) and separates physical pages into disjoint groups based on their colors. Each cache color or physical page group corresponds to a page color, and the physical pages in the same color are mapped to the cache sets in the same color. By manipulating the mapping between virtual pages and physical pages, the page coloring technique can control how the data sets in application virtual spaces are mapped to cache sets, i.e. how the cache space is allocated among the data sets. In the above example, a ULCC supported program changes the layout of its data on physical space, such that threads sharing the same L2 cache visit data on physical pages in different colors, and physical pages for blocks and physical pages for sorting buffers are in different colors.

## 3. Overall Structure and Design of ULCC

ULCC first provides application programmers with an easy-to-use interface to specify how cache space should be allocated among their data sets for efficient use of the last level caches on multicores. Then, in the execution of a program, it enforces a cache space allocation by changing the mapping between virtual pages of the program and physical pages. In this section, we explain how ULCC achieves these objectives. We first introduce the general structure of ULCC. Then we describe its interface and key components.

As shown in Figure 3, in the top layer of ULCC is a set of library functions for programmers to specify desired cache space allocation. We will introduce how an application uses these functions in

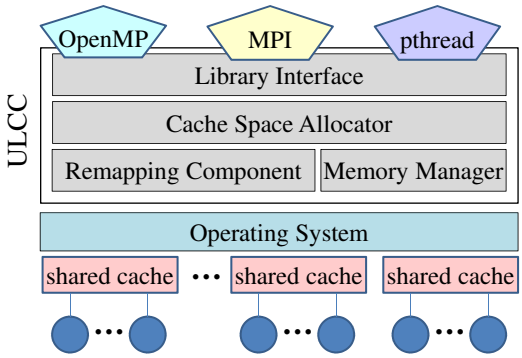


**Figure 3.** The overall structure of ULCC

Section 3.1. The *Cache Space Allocator* (Section 3.2) in the middle layer manages and selects cache colors in the shared caches on a multicore platform to make cache space allocation more efficient. Actual cache space allocation is realized by the *Remapping Component*, which allocates physical pages in the colors selected by the *Cache Space Allocator* and remaps virtual pages over to these physical pages. We will describe this component in Section 3.3. The last component, *Memory Manager* (Section 3.4) reduces the overhead incurred by allocating pages in desired colors. It pre-allocates, manages, and garbage collects physical pages in different colors. When some physical pages in specific colors are required by the *Remapping Component*, the *Memory Manager* can check the physical pages it manages and release the physical pages in the desired colors to satisfy the requirements quickly.

### 3.1 Interface

ULCC provides a number of functions as its interface. With these functions, an application can affect cache space allocation by first defining some cache space slots and then specifying the mapping between data sets and cache slots (i.e. which data set will use which cache slot). To effectively allocate last level cache space among its data sets, the application should specify the cache usage for the data sets in its major data structures (e.g. data sets that account for most accesses). The ULCC interface designs the functions using the following rules.

- ULCC hides most of the complexity of the cache structure on a multicore architecture.

Based on this rule, ULCC allows an application to describe its desired cache allocation in a general way without dealing with machine-dependent details, such as cache colors, associativity, or which cores share the same last level cache. There are three reasons for using this rule: i) Letting programmers explicitly deal with those structural factors unnecessarily increases their burden because the cache structure can be complex on a platform with multiple processors. ii) This rule provides underlying components with more opportunities and space for performance optimization. For example, without specifying cache colors in the program, the *Cache Space Allocator* can have multiple choices on cache colors and choose the ones that can achieve better performance. iii) An application may run on different platforms. Even on the same platform, the amount of available resources for the application may vary across different runs. For example, on a computer with four dual-core processors (each with a shared last level cache), a four-thread application may use two processors or four processors in different runs. Hiding detailed cache structure decouples the application implementation from hardware architecture or configuration variations. Thus application programs can be highly portable.

- The interface allows programs to pass necessary information for underlying components to allocate cache space efficiently. For

example, when ULCC allocates cache space for a data set, it needs to know which threads will access the data set so that it only allocates cache space on the processors running the threads.

- The interface needs to maintain high flexibility for a program to describe how cache space will be allocated among its data sets. Through the interface, a program makes cache space requests, such as how much cache space a data set can use, which data sets can share the same cache space slot and which cannot.

```
/******* MASTER THREAD ********/
...
ULCC_Init(Num_Processors);
/*Available cache space size in KB for the application*/
Space_Avail_App=ULCC_Available_Space_App();
/*Cache space in KB for each thread on average*/
Average_Space=Space_Avail_App/Num_Threads;
...
fork slave threads;
wait for slave threads to complete;
ULCC_Exit();
...
/******* SLAVE THREAD ******/
/*Define a cache space slot*/
Cache_Slot_ID = ULCC_Cache_Slot(Average_Space, private);
/*Define a set of threads*/
TG = ULCC_Thread_Group(NULL, My_Thread_ID);
/*Create a data set to use the cache space slot*/
Data_Set = ULCC_Data_Set(TG);
/*Include data in address ranges from Addr1
to Addr2 and from Addr3 and Addr4 into data set*/
ULCC_More_Data(Data_Set, Addr1, Addr2);
ULCC_More_Data(Data_Set, Addr3, Addr4);
/*Do the actual cache space allocation on the processor
that the thread runs on */
ULCC_Allocate(Data_Set, Cache_Slot_ID);
...
```

**Figure 4.** An example illustrating ULCC function usages

We use Figure 4 to illustrate how a program specifies the desired cache allocation with ULCC functions. The program creates a master thread and a number of slave threads. With ULCC functions, the program evenly partitions the available last level cache space among the slave threads, and in each slave thread it remaps the data set accessed by the thread to the corresponding partition to avoid inter-thread interference in shared caches.

As shown in the *MASTER THREAD* part, the program initiates ULCC by specifying the number of processors the application will run on. The *ULCC_Init* function initializes the ULCC data structures for the application and makes connection to the *Memory Manager*. Then the program gets the size of the available cache space the application can use by calling *ULCC_Available_Space_App*, which calculates the size based on the cache structure of the target platform and the number of processors the application uses. After that, the program calculates the size of the cache space that each slave thread can use.

In the *SLAVE THREAD* part, to allocate cache space, a thread first calls *ULCC_Cache_Slot* to define a cache space slot by specifying the properties of the slot, including the size and whether it is "private" or "shared". "Private" slot is usually for strong locality data. The slot is dedicated to the data sets that are associated with it by the program calling *ULCC_Allocate* with its slot ID (*Cache_Slot_ID* returned by the *ULCC_Cache_Slot* function). When the slot is shared by multiple data sets with strong localities (by calling *ULCC_Allocate* multiple times with the same *Cache_Slot_ID*), the application should guarantee that the data sets time-share the slot without causing much contention (e.g. by visiting data set *A* and then visiting data set *B*). *Cache Space Allocator* in ULCC ensures that "private" cache slots are non-overlapping to secure the cache space for the strong locality data sets. By contrast, a "shared" slot is for weak locality data, and ULCC tries to reuse the cache colors that are already allocated for other weak locality data to make full use of the space in last level caches, as we will explain in the next subsection.

After the cache slot is defined, the thread defines a data set to be mapped to the slot. To define a data set, the thread calls ULCC functions to provide information including which threads access the data set and which data are included in the data set. After that, the thread calls *ULCC_Allocate* to inform the *Cache Space Allocator* to allocate cache colors and *Remapping Component* to change physical pages holding the corresponding virtual pages. If the data set will be accessed by multiple threads (defined in the argument of *ULCC_Data_Set*), ULCC allocates cache space of the specified size on all the processors that the threads run on.

For streaming data with weak spacial locality and randomly-accessed data with weak temporal locality, loading them into caches on accesses causes cache pollution. Making these data non-cacheable can improve performance. Thus ULCC allows a program to mark a data set noncacheable by calling the *ULCC_Allocate* function with the second argument set to *NULL*.

### 3.2 Cache Space Allocator

The *Cache Space Allocator* (briefly *Allocator*) is in charge of managing cache colors. When an application requests some cache space for a data set, the *Allocator* decides which cache colors should be allocated to fulfill the request. The *Allocator* makes the decision in two steps. It first decides how many cache colors should be allocated on each shared cache in the first step. Then it decides which colors should be selected in the second step.

As we have explained in the previous subsection, a program does not prescribe specific colors when it requests cache space. Thus, the *Allocator* has opportunities to improve performance in both steps. In the first step, the *Allocator* aims to maximize cache space utilization and prevent space under-utilization. In the second step, the *Allocator* selects cache colors carefully to reduce the subsequent overhead incurred by ULCC enforcing the cache space allocation.

To efficiently manage and allocate cache colors, the *Allocator* maintains a data structure for bookkeeping the allocation of its colors, for each shared cache on a multicore platform. The data structure marks the status of a cache color to be either *unspecified*, *shared*, or *private*. The cache colors under *unspecified* status are not allocated yet. Cache colors under *shared* status are for weak locality data and can be shared by the weak locality data sets in different threads or even different applications. Because weak locality data lack reuses, the sharing will not cause pollution or contention. Instead, the sharing can minimize the cache space for weak locality, and thus can maximize the cache space for strong locality data to get better performance. The *private* status of cache colors means that the colors are reserved for the exclusive use of some strong locality data sets. Not sharing the colors secures the space for these data sets to avoid cache contention or pollution in these colors. The ULCC interface allows applications to use the "shared" or "private" flag in function *ULCC_Cache_Slot* to denote whether the cache space to be allocated is for weak locality data or strong locality data, and the *Allocator* sets the allocated cache colors accordingly.

In the first step, assuming a cache slot with size $S$ is to be allocated to a data set with size $S_d$ and the size of each cache color is $S_c$, the *Allocator* calculates the number of cache colors that should be allocated on a shared cache, which is $S/S_c$. To prevent under-utilization of cache space, it only allocates the colors in the shared caches on the cores that access the data set, based on the processor affinity information of the threads. For example,

an application will allocate 256KB cache space for a data set. Each cache color is 64KB. If the data set will be accessed by four threads running on two dual-core processors (each with a shared cache), the *Allocator* decides that ULCC allocates 4 cache colors in each of the two shared caches.

If the cache space to be allocated is of "private" type, the *Allocator* continues with the second step to select cache colors. If the cache space to be allocated is of "shared" type, the *Allocator* examines the status of the cache colors in the shared caches. If there are already cache colors under the "shared" status, the *Allocator* will try to "reuse" these colors for the data set. Thus fewer cache colors can be allocated. In the above example, if the application specifies the 256KB cache space as "shared" and there is already a cache color in color 0 having been marked as "shared" in each shared cache, the ULCC reuses the cache color and allocates another 3 cache colors in each shared cache.

In the second step, the *Allocator* carefully selects colors to reduce the subsequent overhead incurred by ULCC enforcing the cache space allocation. To enforce the cache space allocation, the *Remapping Component* (to be discussed in section 3.3) needs to acquire physical pages in the selected colors and use these physical pages to hold the virtual pages of the designated data set. Acquiring the physical pages in a certain color may become more difficult and incur higher overhead when such physical pages become scarce in memory. Thus it is desirable that the *Allocator* carefully selects cache colors to minimize this overhead, especially when the data set is large.

To achieve this objective, ULCC maintains an array to record the number of physical pages in each color that are available in the *Memory Manager*. The array is shared by *Allocator* and *Memory Manager*. When a number of cache colors are to be allocated, the *Allocator* first calculates the average number ($N$) of physical pages that are needed in each color, which is $S_d \times S_c / S$. Then it searches the array and looks for the colors with more physical pages than $N$. The *Allocator* does not select colors with fewer physical pages than $N$. The reason is that if these colors are selected the *Memory Manager* would not satisfy the physical page allocation requirements with existing pre-allocated physical pages and it may take a long time or may even be impossible for the *Memory Manager* to acquire more physical pages. If the cache slot is "private", among the colors with more physical pages than $N$, the *Allocator* selects the noncontinuous colors (e.g. colors 0, 3, 7, 9, instead of colors 0, 1, 2, 3) with the fewest physical pages. The *Allocator* does not select colors with much more physical pages than $N$ because the extra physical pages in these colors become underutilized if there are no other data sets being mapped to the colors. The reason why the *Allocator* chooses noncontinuous colors is to take advantage of the *Memory Manager* to accelerate physical page allocation as we will explain later.

### 3.3 Remapping Component

The *Remapping Component* adjusts the mappings between virtual pages and physical pages to implement actual cache space allocation. It first acquires physical pages in the desired colors corresponding to the allocated cache colors. Then it copies the data set to the newly acquired pages to prevent data loss if the data set has been initialized. Finally, it maps the virtual pages holding the data set to the newly acquired physical pages. To prevent page swapping from changing the physical pages used by the data set, ULCC locks these pages and makes them memory-resident.

To acquire the physical pages in desired colors, the *Remapping Component* first allocates a bunch of physical pages by *malloc*-ing some virtual memory space and writing a byte into each page in the space. Then the *Remapping Component* determines the physical page numbers and the colors of these pages. In the current ULCC implementation under Linux, the *Remapping Component* determines the physical page number of a page through the *pagemap* interface, which allows an application to examine its page table at the user level. The color of a page is equal to the physical page number modulo the number of cache colors of a shared cache. In a system without *pagemap* support, e.g. FreeBSD, Solaris, or Windows, ULCC can use a kernel module or a pseudo-device driver carrying out virtual-to-physical address translation to get physical page numbers. After the *Remapping Component* has selected the pages in the desired colors, it releases unneeded physical pages.

To map the virtual pages holding a data set to the physical pages in the desired colors, the current ULCC implementation makes *mremap()* system calls in Linux system. System call *mremap()* expends or shrinks a memory space, or moves a memory space to another address by changing the mapping between virtual pages and physical pages. On the systems where *mremap()* is not available, ULCC changes the page mappings by creating a shared memory segment and remapping the virtual pages holding the data set to the physical pages in the shared memory segment with *mmap()* system calls. After the virtual pages holding the data set have been remapped over to the physical pages, subsequent accesses to the data set will only use the cache space in the desired colors.

### 3.4 Memory Manager

The *Memory Manager* is a stand-alone process, independent of the applications using ULCC. It pre-allocates, manages, and garbage-collects physical pages to facilitate the allocation of physical pages in the colors required by the *Remapping Component*.

The *Memory Manager* is started as a system service before users run any ULCC-supported applications. When it is started, it first acquires a number of physical pages (e.g. 1/2 of the available physical pages on the system). It organizes the pages into different lists based on their colors, with pages in the same color on the same list. When a ULCC-supported application requires physical pages in specific colors, it notifies the *Memory Manager*. The *Memory Manager* releases some of the physical pages it manages that are in required colors to satisfy the application. Because the *Allocator* usually allocates noncontinuous colors, the released pages are not continuous in physical memory space. Thus the buddy system managing free physical pages in OS kernel puts the physical pages at the head of the free list, which is the place the OS first tries to allocate physical pages from. This makes it much easier for the application to acquire the physical pages in the required colors. After the *Memory Manager* has released the physical pages, the application immediately begins requesting physical pages from the OS and quickly obtains the physical pages the *Memory Manager* just released to satisfy its requirements. ULCC maintains the information on the mappings between data sets and the cache colors. Thus when the memory space holding a data set is released, the ULCC also notifies the *Memory Manager* to garbage-collect the physical pages for future use.

The *Memory Manager* sleeps for most of the time, and is awakened occasionally when a ULCC-supported application requests or releases physical pages. The pages it manages are seldom accessed. To avoid OS page swapping removing these pages, the *Memory Manager* pins them into memory. However, this may increase the memory pressure on the system. To solve this problem, the *Memory Manager* periodically wakes up and checks the amount of free memory in the system. If the amount of free memory is below a threshold, it proactively releases a part of the memory it holds.

## 4. Experiments and Case Studies

In this section, we explore a few case studies and present the experiment results based on the ULCC implementation on a Linux system. We aim to answer the following questions specifically.

- **The usability of ULCC**: Is it easy for a programmer to determine desirable cache allocation based on data locality analysis? Can a programmer easily enforce the cache allocation with the support of ULCC?
- **The performance of ULCC**: How much performance improvement can an application achieve by using ULCC?
- **The overhead of ULCC**: How much overhead does ULCC incur?

### 4.1 Experiment Setup and Workloads

We carried out our experiments on a Dell PowerEdge 1900 workstation with two 2.66GHz quad-core Xeon X5355 processors. Each X5355 processor has two pairs of cores and cores in each pair share a 4MB, 16-way set associative L2 cache. Thus there are 64 cache colors in each shared L2 cache and each cache color corresponds to 64KB of cache space. Each core has a private 32KB L1 instruction cache and a private 32KB L1 data cache. Both adjacent-line prefetching and stride prefetching are enabled on the processors. The workstation has 16GB physical memory with eight 2GB dual-ranked Fully Buffered DIMMs (FB-DIMM). The operating system is 64-bit Red Hat Enterprise Linux AS release 5. The Linux kernel is 2.6.30. The compiler is gcc 4.1.2. The C language library is glibc 2.5. We used pfmon [6] to collect performance statistics such as L2 cache misses.

As case studies, we selected two computational kernels (*MergeSort* [13] and *MatMul* [12]) implemented with pthread and two scientific programs (*CG* and *LU*) from an OpenMP implementation of NAS benchmarks [1]. The *MergeSort* is described in Section 2. The *MatMul* implements a matrix multiplication algorithm similar to that described in paper [12] but with multiple threads. Each program has two implementations. One is the original program that is designed to optimize cache performance for single-core systems with techniques such as blocking and padding. The other is the improved program with ULCC support to reduce cache contention and pollution for its execution on multicore systems. We run each program four times with one thread, two threads, four threads, and eight threads, respectively. Because we were targeting the performance of shared last level caches, for two-thread executions, we used two cores sharing the same L2 cache, and for four-thread executions, we used a pair of cores sharing the same L2 cache on each processor. In each program, we used *sched_setaffinity* to pin each thread to a computing core.

### 4.2 Case Studies

In this section, we use several case studies to illustrate how programmers identify data sets with different localities, determine desired cache space, and use ULCC to enforce cache allocation. We show the performance improvements for both single-thread executions and executions with multiple threads of the selected programs.

- **MergeSort**

In the experiment, we used the program to sort ten million data elements in an array. Each data element has an integer key and a 56-byte content. To avoid severe cache contention, we selected the block size such that the total size of a block and a sorting buffer was equal to the L2 cache size in single-thread execution and equal to half of the L2 cache size in executions with multiple threads.

The major data structures in the program include the original array to be sorted, sorting buffers, the binary trees used by multi-way merging, and the destination array to save final results. Among them, the original array is divided into blocks and data elements in each block are accessed multiple times before they are sorted. Elements in a sorting buffer are accessed multiple times when the program sorts each block until all the blocks have been sorted in each bucket. Nodes in a binary tree are repeatedly used in each multi-way merging. Data sets in these data structures have strong localities. Each element in the destination array is written only once. Thus, the destination array has weak locality. After a block has been sorted, the elements in it are accessed for only one more time in the multi-way merging. Thus elements in it become weak locality data.

To avoid cache contention and cache pollution, the program can protect strong locality data structures with ULCC by securing sufficient cache space for them. Specifically, for the single-thread execution, the program allocates half of the L2 cache space to the original array (i.e. the blocks) and the other half of the L2 cache space to the sorting buffer. When the program is doing multi-way merge, both the original array and destination array have weak locality. Thus they can share the cache space allocated to the original array. Though the binary tree has strong locality, the program lets it share the cache space with the sorting buffer because they are accessed in different execution phases and do not compete with each other for cache space. Similarly, in the execution with multiple threads, for each thread, the program allocates 1/4 of an L2 cache space to the buckets sorted by the thread and the part of destination array saving its sorting results, and allocates another 1/4 of an L2 cache space to the sorting buffer and the binary tree used by the thread.

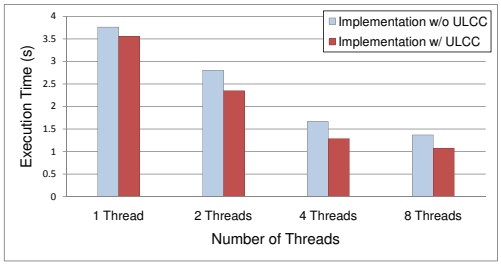


**Figure 5.** The execution times of *MergeSort* implementations with and without ULCC as the number of threads is varied from one to eight.

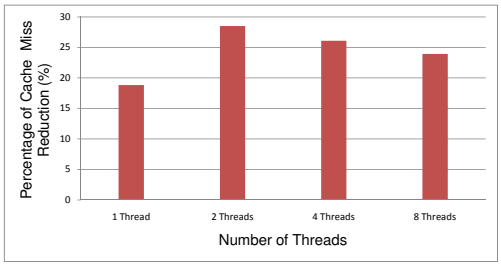


**Figure 6.** The percentages of miss reduction of *MergeSort* with ULCC enforcing the desired cache allocation.

For this program, we show the execution times[1] for both its implementations with and without ULCC in Figure 5 where we vary the number of threads from one to eight. In Figure 6, we show what percentage of L2 cache misses can be reduced by enforcing the above cache allocation. The figures clearly show the effectiveness of delicate user-controlled cache space allocation with ULCC. Even one thread on a single processor can still get performance improvement (execution time reduced by 5.4%) through reducing

[1] We only consider the time spent on sorting the data elements. Thus the time spent on initializing the data array is not included.

L2 cache misses with ULCC optimizing the cache space allocation among its data structures. The reason is that even with a single thread in the program, switching blocks would also evict part of the space previously occupied by the sorting buffer from the L2 cache if the program does not use ULCC to separate the cache space for the original array and the cache space for the sorting buffer. With more threads in the same execution, due to the cache sharing, cache contention and cache pollution further degrades performance. Thus, in these cases cache optimizations supported by ULCC achieve better performance. For example, ULCC reduces the execution time by 22.7% when four threads are used.

- **MatMul**

The *MatMul* program multiplies two double precision matrices $A$ and $B$, and produces the product matrix $C$. The size of each matrix is 2048 $\times$ 2048. To achieve necessary data reuses in L2 caches, the matrix multiplication is carried out block by block. For the block $a$ on the $i$th block row and $j$th block column of matrix $A$, it is multiplied with all the blocks on the $j$th block row of matrix $B$, and the results are accumulated into the blocks on the $i$th block row of matrix $C$. Before the program finishes the computation with block $a$, it is desirable that the data in $a$ can be kept in the cache. However, without a dedicated space for block $a$, the data in it may be repeatedly evicted from the cache before its next use every time the program switches blocks in matrix $B$ and matrix $C$, even with a rather small block size. To reduce the chance that the data in each block of matrix $A$ is evicted from the last level cache prematurely, we set the block size to 360 $\times$ 360 (about 1MB) for the single-thread executions and 256$\times$256 (512KB) for other executions.

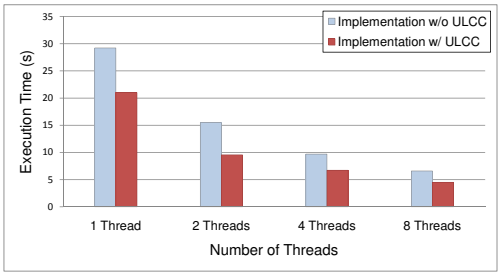


**Figure 7.** The execution times of *MatMul* implementations with and without ULCC as the number of threads is varied from one to eight.

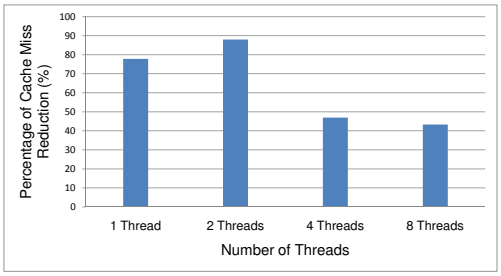


**Figure 8.** The percentages of miss reduction of *MatMul* with ULCC enforcing the desired cache allocation.

Figure 7 compares the execution times of the original implementation and the implementation that allocates dedicated cache space for the blocks in $A$ leveraging the ULCC support. Though the block size is much smaller than the L2 cache size, the extra cache misses caused by data in blocks of matrix $A$ being evicted from L2 caches can still degrade the performance of *MatMul* by over 30%. With ULCC, the program allocates a 1MB cache space to matrix $A$, and allocates the remaining cache space to matrices $B$ and $C$, for the single-thread execution. In the cases with multi-thread executions, for each thread, the program allocates a 512KB cache space to the part of matrix $A$ it uses. The remaining cache space is shared by matrices $B$ and $C$. With the dedicated cache space, each block in $A$ is protected after it is loaded into an L2 cache. This significantly reduces L2 cache misses, as shown in Figure 8, and avoids performance degradation.

- **NAS LU**

NAS LU benchmark uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular-block upper triangular system resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions by splitting it into block lower and upper triangular systems. In the experiments, we run the benchmark with input class A.

Among the major data structures of the application, the data structures for the three dimensional field variables and residuals (arrays *frct*, *flux*, *u*, and *rsd*) have weak locality. They are referenced with a looping access pattern, and their sizes far exceed the L2 cache size. In each time-step iteration, they are accessed only a few times. By contrast, arrays *a*, *b*, *c*, and *d* have strong locality. Their sizes are relatively small and fit into an L2 cache. In each time-step iteration, they are repeatedly accessed multiple times for each $z$ plane. Accessing the data structures for the field variables and the residuals causes cache pollution because it flushes the data in arrays *a*, *b*, *c*, and *d* from L2 caches. To avoid cache pollution, with ULCC, the program allocates dedicated L2 cache space to arrays *a*, *b*, *c*, and *d*. This also keeps the arrays away from interference incurred by the application accessing other data structures.

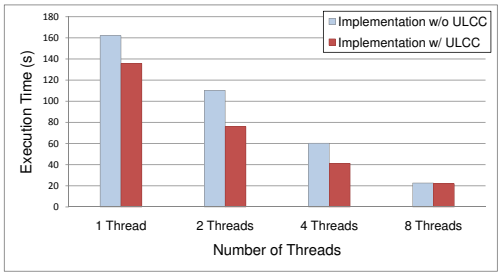


**Figure 9.** The execution times of *LU* implementations with and without ULCC as the number of threads is varied from one to eight.

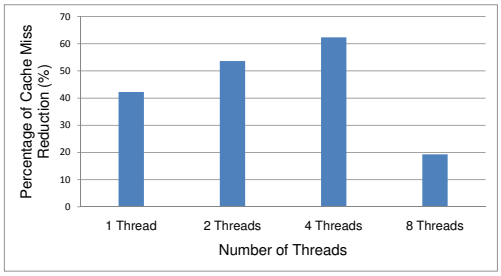


**Figure 10.** The percentages of miss reduction for *LU* with ULCC enforcing the desired cache allocation.

Figure 9 shows the execution times for the implementation without ULCC and the implementation with ULCC as the number of threads is varied from one to eight. Figure 10 shows the percentages of L2 cache miss reduction through the cache optimization. For the execution with two threads and the execution with four threads, the cache optimization supported by ULCC reduces L2 cache misses

by 54% and 62% respectively, and reduces the execution times by 31% and 32% respectively.

For the execution with eight threads, cache optimization cannot achieve as much performance improvement as it does for the executions with fewer threads. This is because arrays $a$, $b$, $c$, and $d$ are split among the threads in *LU*, and each thread only accesses a portion of elements in each array. In the eight-thread execution, each thread accesses a smaller portion of the data in these arrays than it does in the executions with fewer threads. Consequently, data elements in these arrays become less likely to be evicted from L2 caches by the accesses to weak locality data sets in the eight-thread execution than they do in the executions with fewer threads.

For single-thread execution, the cache optimization helps to reduce the execution time by 16%, which is less than that of the execution with two threads. This is because cache pollution with a single thread is not as intensive as that with two threads sharing the same L2 cache.

- **NAS CG**

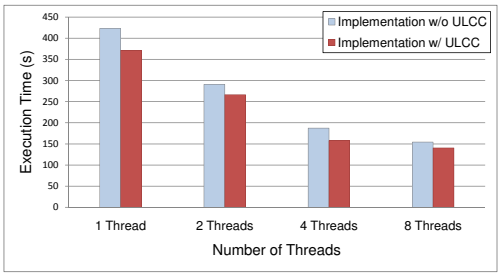


**Figure 11.** The execution times of *CG* implementations with and without ULCC as the number of threads is varied from one to eight.

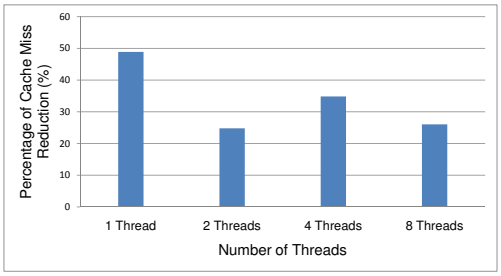


**Figure 12.** The percentages of miss reduction for *CG* program with ULCC enforcing the desired cache allocation.

The CG benchmark spends most of its execution time on multiplying a sparse matrix and a dense vector. The non-zero elements in the matrix are stored in array $a$, and the vector is stored in array $p$. The program uses another array $colidx$ to hold the column indexes for the non-zero elements in the matrix. Thus, for each element $a[k]$ in $a$, there is an element $colidx[k]$ in $colidx$ storing its column number in the matrix.

When the program calculates the product vector, for each non-zero element $a[k]$ in the matrix, the program visits array $colidx$ to get the column index $colidx[k]$ of the element, and multiplies $a[k]$ with the $colidx[k]$-th element in the vector, i.e. $p[colidx[k]]$. Thus, during the matrix-vector multiplication, elements in $a$ and $colidx$ are visited only once. However, elements in $p$ are repeatedly visited because the matrix have multiple rows. Obviously, $a$ and $colidx$ are weak locality data structures, and $p$ is a strong locality data structure.

The implementation with ULCC limits the cache space for $a$ and $colidx$ by allocating only a minimum number of cache colors to these arrays. Thus accessing these arrays will not evict the data in array $p$ from L2 caches. As shown in Figure 11 and Figure 12, the above cache optimization reduces L2 cache misses by 25% to 50%, and reduces the execution times by 8% to 16% accordingly (input class is C). Among the executions with different number of threads, cache optimization reduces the number of cache misses in the single-thread execution by the largest percentage, but it cannot reduce the execution time by the largest percentage. This is because in a single-thread execution the L2 cache space is dedicated to one thread. Thus the miss rate of a single-thread execution is lower than that of multi-thread executions, and reducing cache misses does not have as much impact on execution time as it does for multi-thread executions.

### 4.3 Experiments to Measure Overhead

To optimize cache performance with ULCC support, an application has to pay some overhead for acquiring physical pages in desired colors, copying data, and changing the mapping between virtual pages and physical pages. In this subsection, we measure the overhead. We show that with the *Memory Manager*, the overhead can be significantly reduced to a level comparable to that of memory allocation, which is negligible for scientific applications.

We used a micro-benchmark to measure the overhead. The benchmark allocates a cache slot to a 32MB data set with ULCC functions and measures the time. We run the benchmark multiple times, each time with a different cache slot size selected from 64KB to 2MB. Figure 13 shows the times measured with the benchmark. To highlight the benefit of using *Memory Manager*, we show the times for the ULCC implementations with and without *Memory Manager*.

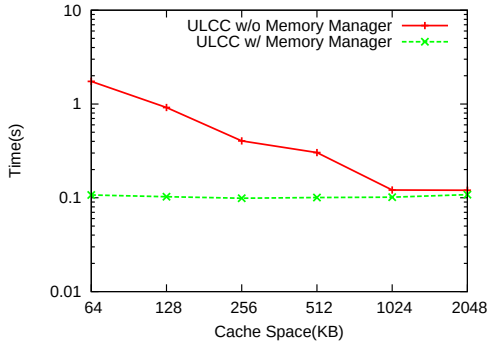


**Figure 13.** The overhead of ULCC implementations with and without *Memory Manager*. Both axises are in log scale.

As Figure 13 shows, without *Memory Manager*, ULCC may incur some overhead that is non-negligible for short-running applications. For example, it takes nearly 2 seconds to allocate a 64KB cache space to the 32MB data set. The main reason is that without the *Memory Manager*, the *Remapping Component* must spend long time to acquire a large number of physical pages, from which it selects physical pages in the required colors to finish remapping. For example, when ULCC allocates 64KB cache space in a selected cache color for the data set, it needs to acquire and examine $32MB \times 64 = 2GB$ of physical memory (64 is the total number of cache colors in a shared L2 cache) on average to fulfill its requirements (i.e. 32MB physical memory in the selected color). According to our measurement, acquiring 2GB physical memory takes about 1.6 seconds, while copying the data set into the selected 32MB physical space takes only 0.02 second and adjusting the mapping between the corresponding virtual pages and the physical pages takes only 0.03 second. Fewer physical pages are needed when the benchmark allocates larger cache space to the same data

set. For example, if 512KB cache space (corresponding to 8 cache colors) is being allocated to the data set, only 256MB physical memory on average is needed to get 32MB physical memory in the required colors. which takes much less time. Thus, a trend we have observed is that the time spent on allocating cache space for the data set decreases significantly when the size of the cache space increases.

The *Memory Manager* can reduce overhead significantly, especially when the size of the allocated cache space is small. The reason is that with *Memory Manager*, a ULCC-supported application does not need to acquire so much physical memory to obtain enough physical pages in desired colors. With the pre-allocated physical pages and the knowledge of their colors, the *Memory Manager* can "pass" physical pages in the required colors to the *Remapping Component* without the overhead incurred by acquiring extra physical pages. With the *Memory Manager*, the *Remapping Component* can also reduce the overhead incurred by page remapping. Without *Memory Manager*, the physical pages are mapped one by one to the virtual space of the data set, and mapping each physical page requires a *mremap()* call. By contrast, with the *Memory Manager*, most physical pages obtained by the *Remapping Component* are continuous in virtual space. The physical pages continuous in virtual space can be mapped together with one *mremap* call.

To compare ULCC with the design from direct OS kernel support, we also modified the Linux kernel to allocate free physical pages in the colors required by user level applications. We found that ULCC with *Memory Manager* incurs comparable overhead to that of OS kernel support. The overhead is also comparable to that of allocating physical pages regardless of their colors and copying the data set to the newly allocated pages. For example, it takes ULCC 0.1 second to allocate 256KB cache space to a 32MB data set, while it takes Linux operating system 0.03 second to allocate 32MB memory regardless of page colors, and it takes 0.02 second to copy 32MB data to the newly allocated memory space. These experiments show that the overhead incurred by ULCC is reasonable, especially for scientific applications whose execution times are usually long.

In most cases, a programmer can specify the desired cache allocation at the beginning of the program when the data structures in the program have not been initialized or allocated with physical space. Thus when the program allocates cache space to the data structures, there is no need to copy any data over to the newly allocated physical space. At the same time, because the physical space for the data structures has been allocated by ULCC, there is no need for OS to allocate physical space when the data structures are being initialized. Thus ULCC incurs much less overhead in these cases.

## 5. Related Work

Our work is related to the following research areas: cache conscious program design and library design to improve the productivity of parallel programming and efficiency of parallel programs, cache partitioning to provide each of the running threads with a chunk of dedicated cache space to avoid interference from other co-running threads, and sophisticated scheduling policies for multicore or SMT (Simultaneous MultiThreading) processors to co-schedule threads that can efficiently use the shared resources in multicore/SMT processors.

### 5.1 Cache Conscious Program Design and Library Design

As a major method to improve application performance in high performance computing, restructuring algorithms and programs to make an efficient use of CPU caches has been intensively studied [5, 12, 20, 28, 30, 32]. Common techniques exploit data access locality by rearranging computing operations to improve temporal locality, such as strip mining [28], loop interchange, blocking [12], or reorganizing data layout for better spacial locality [5]. Libraries that implement dense linear algebra operations in a cache conscious way, such as BLAS [3], LAPACK [2], and ATLAS [29], have been extensively used for performance programming.

However, the approaches above are designed for single-core systems, and cannot address the cache contention and pollution problems in shared caches on multicore processors. Addressing these problems requires a high degree of synergy between the programs themselves and different system components. ULCC provides programmers with an easy-to-use interface to address the problems effectively, while it hides and takes over the tasks of dealing with complex structures and operations from the users.

There are other libraries or runtime systems that facilitate parallel programming, such as PVM and MPI. However, they don't handle cache optimizations. ULCC can be an important supplement to them.

### 5.2 Cache Partitioning

Cache partitioning has been confirmed to be an effective approach to address cache contention and pollution problems. Various hardware cache partitioning solutions have been proposed, which allocate a chunk of dedicated cache space (partition) for each running thread and dynamically adjust the sizes of the partitions according to the cache requirements of the threads [10, 17, 21, 24]. Due to the extra complexity and chip overhead to implement cache partitioning in hardware, hardware supported cache partitioning has not been available in commercial multicore processors. Thus several studies use software approaches to separate the cache space used by different running threads with the page coloring technique in operating systems [14–16, 25, 34]. Recent research results have confirmed that separating cache space used by data with different localities reduces cache misses because it avoids cache pollution incurred by weak locality data [18, 23].

However, there are no facilities for programmers to use to control directly the cache space allocation among the data structures in a program. Existing work focuses on automatically detecting access patterns with profiling or with hardware support, and lacks enough flexibility needed by programmers to leverage their insightful understanding of the program to enforce the cache allocation they desire. In addition, existing work focuses on single node system and relies on heavy modification of operating system key components. Their solutions cannot be portably used in existing high performance computing environments. Thus a portable and flexible facility like ULCC addresses this concern to enable programmers to exploit fully the benefits from optimizing cache space allocation.

### 5.3 Multicore/SMT-aware Scheduling

Threads co-running on an SMT or a multicore processor share and compete for the shared resources on the processor such as functional units, shared caches, and memory bus. How the shared resources can be efficiently used becomes a critical issue. As one of the major approaches to address this issue, scheduling has been focused on by a large number of papers [4, 7, 11, 22, 26, 35].

These papers focus on general systems such as desktop systems and server systems where scheduling plays an important role. However, scientific applications usually require only basic scheduling functions, and in most cases, for better performance scientific applications statically map threads to computing cores by setting their affinities.

## 6. Conclusion

We have proposed and implemented a user level facility called ULCC that enables programmers to control explicitly cache space

allocation among the data structures in their programs to maximize the utilization of shared last level caches on multicore processors by reducing cache pollution and cache contention. ULCC provides an easy-to-use interface for programmers to specify the desired cache allocation without having efforts to deal with complex structures and operations in the memory hierarchies on the systems with multiple multicore processors. In ULCC, the *Cache Space Allocator* efficiently allocates cache colors to fulfill the requirements of a program. The *Remapping Component* enforces the desired cache space allocation by adjusting the mapping between virtual pages and physical pages with existing operating system support. Thus ULCC does not require modifications in operating system kernels. To minimize the overhead of ULCC, the *Memory Manager* preallocates, manages, and garbage-collects physical pages, so that the *Remapping Component* can acquire physical pages in desired colors and adjust the mappings quickly.

The benefits to users for achieving high performance and high throughputs may come from ULCC in two ways. First, ULCC-based programming secures sufficient cache space for data structures with strong locality, and limits cache space for data structures with weak locality. Second, ULCC can further help users to adjust block sizes to maximize throughputs. Our experiments have shown the effectiveness of ULCC with various applications. We will further refine and optimize ULCC for its wide usage in application communities.

## 7. Acknowledgments

The authors thank the anonymous reviewers for their constructive comments and suggestions. They thank Bill Bynum for reading the paper and for his suggestions. They also thank Hao Wang, Feng Chen, and Rubao Lee for helpful discussion. This research was supported by National Science Foundation under grants CNS0834393 and CCF0913150.

## References

[1] NAS parallel benchmarks in OpenMP. URL `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *SC '90*, pages 2–11, 1990.

[3] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[4] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA'07*, pages 105–115, 2007.

[5] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI '99*, pages 1–12, 1999.

[6] HP Corp. Perfmon project. URL `http://www.hpl.hp.com/research/linux/perfmon`.

[7] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT'08*, pages 220–229, 2008.

[8] D. Kang. Dynamic data layouts for cache-conscious factorization of DFT. In *IPDPS '00*, page 693, 2000.

[9] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4), 1992.

[10] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT'04*, pages 111–122, 2004.

[11] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[12] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS '91*, pages 63–74, 1991.

[13] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *SODA '97*, pages 370–379.

[14] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: minimizing cache conflicts in muli-core processors for databases. In *VLDB'09*.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA '08*, pages 367–378, Salt Lake City, UT, 2008.

[16] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Enabling software multicore cache management with lightweight hardware support. In *SC'09*, 2009.

[17] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA'04*, pages 176–185, 2004.

[18] Q. Lu, J. Lin, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *PACT '09*, pages 246–257, 2009.

[19] S. K. Moore. Multicore is bad news for supercomputers. pages 213–226, 2008.

[20] M. Penner and V. K. Prasanna. Cache-friendly implementations of transitive closure. In *PACT '01*, page 185, Barcelona, Spain, 2001.

[21] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO'06*, pages 423–432, 2006.

[22] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS'02*, pages 66–76.

[23] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *MICRO '08*, pages 258–269, 2008.

[24] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomputing*, 28(1), 2002.

[25] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *WIOSCA '07*, 2007.

[26] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys'07*, pages 47–58, 2007.

[27] TOP500.Org. URL `http://www.top500.org/lists/2010/06`.

[28] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip mining redistribution. In *PARLE '94*, pages 323–335, 1994.

[29] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SC '98*, 1998.

[30] M. Wolfe. Iteration space tiling for memory hierarchies. In *PP '89*, pages 357–361, Philadelphia, PA, 1989.

[31] M. Wolfe. More iteration space tiling. In *SC '89*, pages 655–664, 1989.

[32] L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM J. Exp. Algorithmics*, 5:2000, 2000.

[33] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA '07*, pages 93–104, 2007.

[34] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *EuroSys'09*, pages 89–102, 2009.

[35] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10*, pages 129–142, 2010.