Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2011-032 June 28, 2011

A Software Approach to Unifying Multicore Caches

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

A Software Approach to Unifying Multicore Caches

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich

Abstract

Multicore chips will have large amounts of fast on-chip cache memory, along with relatively slow DRAM interfaces. The on-chip cache memory, however, will be fragmented and spread over the chip; this distributed arrangement is hard for certain kinds of applications to exploit efficiently, and can lead to needless slow DRAM accesses. First, data accessed from many cores may be duplicated in many caches, reducing the amount of distinct data cached. Second, data in a cache distant from the accessing core may be slow to fetch via the cache coherence protocol. Third, software on each core can only allocate space in the small fraction of total cache memory that is local to that core.

A new approach called software cache unification (SCU) addresses these challenges for applications that would be better served by a large shared cache. SCU chooses the on-chip cache in which to cache each item of data. As an application thread reads data items, SCU moves the thread to the core whose on-chip cache contains each item. This allows the thread to read the data quickly if it is already on-chip; if it is not, moving the thread causes the data to be loaded into the chosen on-chip cache.

A new file cache for Linux, called MFC, uses SCU to improve performance of file-intensive applications, such as Unix file utilities. An evaluation on a 16-core AMD Opteron machine shows that MFC improves the throughput of file utilities by a factor of 1.6. Experiments with a platform that emulates future machines with less DRAM throughput per core shows that MFC will provide benefit to a growing range of applications.

1. Introduction

Multicore systems have increasingly distributed caches. There is an inherent cost to accessing far-away cache memory, both between chips on a motherboard, and within a single chip. To make cache memory fast, architects place cache units closer to each individual core. In theory, this provides a large amount of aggregate cache space across the system, and fast cache memory for each individual core. Unfortunately, it can be difficult for an application to use such a distributed cache effectively. This paper explores software techniques that help applications make better use of distributed caches on multicore processors.

Consider the difficulties faced by an application trying to use a distributed cache on a multicore processor. In some processors, an application can cache data only in the cache memory associated with its execution core. This provides the application with access to only a fraction of the overall system's cache capacity, and cache memory on other cores remains unused. Even if the application were to execute code on many cores in parallel, each core's individual caches would end up caching the same commonly accessed data, leaving the number of distinct cached items—and thus, the effective capacity—low compared to the total on-chip cache size.

Other architectures try to address this problem by allowing applications to cache data in either far-away or shared caches. While this gives the application access to more cache capacity, the slower access times of a far-away or shared cache can reduce the effectiveness of caching.

This paper explores the opportunity for better management of distributed caches in multicore processors. We first show that current off-the-shelf multicore processors can benefit from better cache management using a technique called *software cache unification*, or SCU. We then simulate future multicore processors, which have less DRAM bandwidth per core, and show that SCU will become more important over time.

SCU allows applications to make better use of distributed caches, by combining the speed of local caches with the large aggregate capacity of a shared cache. In order to use all of the available cache space, SCU explicitly manages the assignment of application data to individual cache memories that are part of the system. Then, when a particular thread is about to access a piece of data, SCU migrates the thread to a core near the cache assigned to that data item. Thus, SCU allows applications to use the large aggregate cache capacity of multicore processors while avoiding costly accesses to far-away caches.

The key to SCU's operation is thread migration, which serves two distinct functions. First, thread migration helps SCU control cache placement. When a thread references a data item, that thread's core fetches the data into its cache. Thus, by migrating the thread to a particular core, SCU can control which core's cache will hold that data. This allows SCU to limit how many times the data is duplicated in on-chip caches. Second, thread migration helps SCU reduce the cost of accessing a far-away cache, by executing the thread on a core closer to the needed data.

A significant challenge in implementing SCU is balancing the cost of migration with the benefits of larger cache capacity or with the costs of accessing a far-away cache. If SCU spends too much time migrating threads, it may reduce application performance, even if it improves caching efficiency. Another challenge revolves around deciding when to evict unused data from caches, or when to replicate a popular data item in more than one cache, without low-level access to the eviction hardware. Finally, modern hardware has complex cache eviction algorithms, which make it difficult for SCU to predict what data is stored in which cache at any given time. Thus, SCU must be robust in the face of partial or even incorrect caching information.

To demonstrate the feasibility of SCU, we have modified the Linux file cache to use our approach. The resulting system, called MFC (short for *Multicore-aware File Cache*), transparently migrates unmodified Linux applications between cores as they access file data using traditional `read` and `write` system calls. Applications that want more control over cache placement, or use memory-mapped files, can use an explicit interface to influence MFC. MFC implements migration by removing a thread from its current per-core Linux run queue and adding it to the target core's queue.

We evaluated the MFC implementation on a 16-core AMD platform running applications and microbenchmarks. MFC improves performance of unmodified file utilities by up to a factor of 1.6 compared to running on Linux without MFC, and improves performance even for working sets much larger than the total on-chip cache. On a simulated future multicore chip, which has increasingly more cores without a corresponding increase in the number of DRAM channels, MFC performs even better. The microbenchmarks demonstrate that MFC's heuristics do a good job approximating MFC's goal,

never worse than running without MFC, and that MFC improves performance because it reduces the number of DRAM loads.

The main contributions of this paper are: (1) the idea of SCU to unify multicore caches; (2) MFC, which uses SCU to improve the performance of file-intensive applications; (3) an evaluation of MFC with several applications that demonstrates that SCU provides benefits on today's multicore platforms; and (4) an exploration of the benefits of MFC on emulated future multicore platforms. MFC demonstrates that the idea of SCU is effective for improving the performance of read-intensive file-system applications, but we believe that the idea of SCU is also more generally applicable. With the right hardware support SCU could complement and simplify the design of globally-shared caches, making it applicable to a wider range of applications.

The rest of this paper is organized as follows. §2 details the memory challenges caused by multicore processors. §3 explains our approach to dealing with distributed caches. §4 describes the MFC design. §5 summarizes the salient details of our MFC implementation for Linux. §6 measures the benefits of MFC on the AMD platform. §7 speculates how the idea of SCU could be made more broadly applicable. §8 puts the MFC design in the context of related work. §9 summarizes our conclusions.

2. Multicore Caching Challenges

This section outlines the challenges that SCU addresses, which largely spring from the distributed nature of the on-chip cache memory on multicore chips. The main focus of this discussion is read-only or read-mostly data; that is, the focus is on cache capacity rather than invalidations.

To be concrete, we will illustrate these challenges in the context of a specific machine, consisting of four quad-core 2 GHz AMD Opteron chips, although the underlying problems are applicable to other systems. Figure 1 summarizes the memory system of this machine. Each core has its own 64KB L1 and 512KB L2 caches (mutually-exclusive), and the four cores on each chip share a 2 MB L3 cache. A core can access its own L1 cache in three cycles, but requires 50 cycles to access the shared L3 cache, and 121 cycles for a different core's L1 cache. Access to DRAM, or to caches on a different chip, takes at least 200 cycles. Each chip has two DRAM controllers, for a total of 8 in the entire machine.

The four-chip AMD system has a mixture of private and shared caching. Each core's L1 and L2 cache is private to that core: only that core can allocate space in those caches, and the L2 caches of different cores can independently cache copies of the same data. Each chip's L3 cache is shared among the cores on that chip, in that data used by all those cores might be stored just once in the L3 cache. On the other hand, each chip's L3 cache is private to that chip: only cores on that chip can allocate space in the L3, and the L3 caches of different chips can independently cache copies of the same data.

At a high level one can view parallel software running on the 16 cores as having two kinds of read-only data that interacts with the caches: private data used by just one core, and shared data used by many cores. A core can store its private data in its own L1 and L2 caches, and in a fraction of the local chip's L3 cache, depending on other uses of the L3 cache. Even in the best case, when no other cores are actively using cache space, software running on a core can cache at most 2.5MB of private data, out of the full 16MB available. SCU can help avoid this limitation.

For shared data, in the extreme case of all 16 cores using entirely the same data, the same cached data is likely to be stored in the caches of different chips. Within each chip, each L2 cache is likely to store the same data as the other L2 caches. Thus, the amount of distinct data stored in the on-chip caches will be about 2.5MB, far short of the full 16MB aggregate cache capacity. SCU can help applications use cache capacity more effectively by avoiding duplication of shared data.

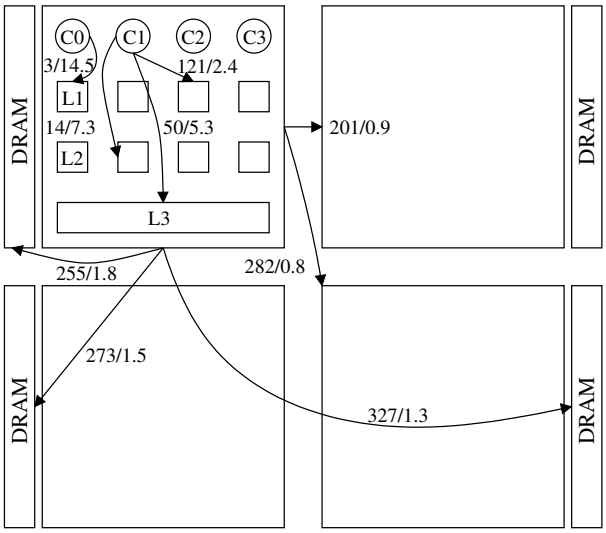


Figure 1. The AMD 16-core system topology. Memory access latency is in cycles and listed before the slash. Memory bandwidth is in bytes per cycle and listed after the slash. The measurements reflect the latency and bandwidth achieved by a core issuing load instructions. The costs of accessing the L1 or L2 caches of a different core on the same chip are the same. The costs of accessing any cache on a different chip are the same. Each cache line is 64 bytes, L1 caches are 64 Kbytes 8-way set associative, L2 caches are 512 Kbytes 16-way set associative, and L3 caches are 2 Mbytes 32-way set associative.

Finally, for both kinds of data, the hardware cache replacement policy is basically to evict the least-recently-used line from the associativity set of the newly installed line. This replacement policy is fairly local; it does not consider other associativity sets in the same cache, or the contents of other caches. This often leads to eviction of data that is far from globally least-used. SCU helps make the global replacement policy more optimal.

It seems likely that future multicore chips will dedicate part of the total on-chip cache to private caches, and thus potentially provide less cache capacity to some applications than would be possible with a fully shared cache. One possible future might look like the 64-core Tilera TILE64 [25], in which all cache space is private. Applications running on such chips could benefit from the larger cache capacity offered by SCU.

At the other end of the spectrum, the Intel 32-core x86 Larrabee [17] graphics processor has a shared cache that is physically partitioned by address into a piece next to each core. A core can use data from any cache partition, but has fastest access to the local partition. In this case, applications already have access to the entire cache capacity, but would still benefit from SCU by accessing nearby cache partitions as opposed to far-away ones.

The techniques for efficient use of the entire on-chip cache capacity explored in this paper are likely to become more important as the number of cores per chip increases. More cores will lead to more fragmentation of the total cache space, and thus more opportunity for duplication and consequent misses for data displaced by those duplicates. More cores will also lead to longer queues of misses waiting to be served by the off-chip memory interface, and thus higher memory latencies.

3. Approach: Software Cache Unification

As §2 explained, current multicore chips typically provide a mixture of shared and per-core private caches. A pure private arrangement might provide the best performance for software workloads in which the working set of data needed on each core can fit in one core's share of the overall cache memory. A shared cache may be best when multiple cores use the same data, and the data is too large to copy into each core's cache. In such cases the single shared cache's high hit rate outweighs its relatively slower hit latency.

The high-level goal of SCU is to provide many of the good properties of a single large shared cache from a multicore chip's private caches. It is aimed at applications whose running time is dominated by memory access (rather than by computation), and which have access patterns that could benefit from a shared cache with size equal to the total on-chip cache. For simplicity, the following discussion assumes that all of the on-chip cache takes the form of private caches, though SCU is also applicable in mixed private/shared designs. In a mixed design, "a core's cache" denotes all cache levels associated with that core as well as a hardware-determined portion of any shared caches. For the AMD system in Figure 1, "a core's cache" means the associated L1 and L2 caches and a portion of the shared L3 cache.

SCU uses software techniques to manage on-chip cache memory. It tracks the popularity of different data items (files, in the case of MFC), and arranges to keep only the most popular items in on-chip cache; that is, it provides a global replacement strategy. It decides how many copies of each data item to store on-chip: this number can be greater than one for very popular items that can benefit from concurrent access on multiple cores. In general SCU arranges to keep at most one copy of each data item in on-chip cache, to maximize the amount of distinct data on-chip and thus minimize the number of misses to DRAM.

SCU must take explicit measures to achieve the above properties, since none of them hold for straightforward use of a multicore chip with private caches. The main technique SCU uses to manage the caches is computation migration. SCU migrates a computation to a particular core in order to ensure that the data the computation uses will be cached on that core. That is, if SCU's partition of the data over the cores indicates an item is on a particular core, then SCU will migrate a thread to that core just before the thread uses the corresponding item. If the item was not previously cached, accessing the item on that core will cause it to be cached there. SCU will migrate future accesses of that data to the same core, so that the data will stay cached in just one core's cache. For data that SCU decides is too unpopular to cache at all (i.e. the amount of more-popular data is enough to fill the total on-chip cache), SCU arranges to access it using non-caching loads so that it will not displace more valuable data.

Only certain kinds of applications will benefit from SCU. The application must be data-intensive as opposed to compute-intensive. (We expect that many future parallel applications will be data-intensive because they will be parallel versions of existing serial data-intensive applications and won't be able to keep all cores 100% of the time busy computing.) The application must also use the same data multiple times, so that caching has a possibility of being effective. The application must actively use only one data item at a time—that is, the application must not switch from one item to another so quickly that the migration cost is prohibitive. The application's working set (i.e., a reasonable fraction of the popular data) must fit in the total on-chip cache, but it must not be so small that it can be replicated in every private cache.

In the best case, SCU can increase the total amount of cached data available to each thread from one core's cache's worth to an effective cache size equal to the total cache in the entire system. How much that improves application performance depends on how its miss rate decreases with cache size, on what fraction of its time it spends reading memory, on the hardware costs of cache accesses and RAM accesses, and on the cost of SCU's migration. SCU also helps performance by migrating computations close to the data they use, converting cross-chip data movement to local cache accesses.

4. Multicore-aware file cache (MFC)

To explore whether SCU is viable, this section describes the design of MFC, a multicore-aware file cache that uses SCU. Applications that work with files, such as file utilities like `grep` or `wc`, or Web servers serving static files, are good examples of data-intensive applications that could benefit from SCU. Furthermore, files are likely to be large enough to recoup the cost of migrating a thread, the dominant cost of SCU. MFC uses the idea of SCU to keep the most-frequently-used files in the on-chip distributed caches and to provide threads with fast access to those files, and, as a result, can improve the performance of file applications.

To achieve this goal, MFC extends an existing file cache by intercepting the call to read a file (i.e., the `read` system call in Linux). MFC must perform four main tasks when a thread reads a file: 1) decide which cache on the chip should hold the file if none already does; 2) decide which file to remove from the caches to make room for the new file; 3) load the new file into the chosen cache; and 4) migrate the thread to the core with the chosen cache.

This section explains the main challenges MFC faces in performing these 4 tasks, describes the approach to addressing them, outlines MFC's cache placement algorithm, and details specific aspects of this algorithm.

4.1 Challenges and approach

To appreciate the challenge of designing a MFC that holds the most frequently-used files, consider a workload that a Web server might generate. A typical Web server has a workload that follows a Zipfian distribution [1]: a few popular files receive most of the requests (e.g., 50%) and a large number of individually unpopular files together receive the remaining requests. Ideally, as many popular files as possible would be cached on-chip, but the way that the cache memory is fragmented and distributed over the chip makes this a difficult goal for the following reasons.

First, as we have described earlier, software running on a core has fast access only to nearby caches, can cause data to be loaded only into local cache, and runs the risk of limiting the amount of distinct data cached if it accesses data already cached on another core. It is for these reasons that MFC migrates threads among cores.

Second, MFC must balance its use of the computation and cache resources of each core. For example, if a file is very popular, the core whose cache holds that file may receive many migrating threads and become a CPU bottleneck. MFC copies such files to multiple cores' caches so that enough compute resources are available.

Third, MFC has no direct control over the hardware's cache replacement algorithm, and the hardware may in any case lack a global replacement strategy. MFC needs a way to allow access to unpopular files, but to prevent the hardware from letting those files evict popular data. The presented design for MFC assumes that the processor has non-caching loads, although we also have a more complex design that works in the absence of non-caching loads.

Fourth, migrating a thread from one core to another core takes time. For small or popular files it may be more efficient to replicate the file data in the cache of the thread's current core, rather than to migrate the thread to the core that's currently caching the file data.

Fifth, MFC is not the only user of the hardware caches; other code and non-file data are cached in unknown quantities. MFC needs to estimate a reasonable amount of cache space for it to use.

4.2 MFC's algorithm

MFC selects which file data to cache by monitoring `read` system calls. When a thread invokes `read` on the core *curcore*, the kernel first calls the `mfc_access` function shown in Figure 2. This function decides what core should read the file, and returns that core as *newcore*. It also returns an indication of whether the file should be read using caching or non-caching load instructions. If *newcore* is different from *curcore*, the kernel migrates the thread to *newcore* before proceeding with the `read`. `read` copies the file from the kernel file cache to user memory either using caching or non-caching loads. After `read` completes, the kernel leaves the thread on *newcore*.

MFC's use of migration has two effects. If the file is not already in *newcore*'s caches, migrating the `read` and using caching loads will cause it to be. If the file is already in *newcore*'s caches, `read`'s copy to user space will run quickly. In either case the user-space copy will be in *newcore*'s caches, where the thread has fast access to them.

`mfc_access` maintains some per-file information. It maintains in `f.cores` a list of the cores in which it believes file f is cached. A frequently-used file may be cached on more than one core to allow parallel reads. MFC also maintains `f.use[a]`, which counts, for each file f and each core a in which the file is cached, the number of times that cached copy has been used. We now briefly describe the MFC decision function; subsequent sections provide more detail.

On lines 3–4, `mfc_access` first checks the file size, and does not migrate for small files. This avoids situations in which the migration overhead is larger than the cost of fetching the file to the local cache. Fetching the file may cause it to be duplicated in multiple caches. MFC also treats very large files specially (see §4.7), though the pseudo-code omits this case.

Lines 6–10 check if *curcore* has spare capacity in its cache (and *curcore* is a frequent user of f, not reflected in pseudo-code), and if so, MFC replicates f on *curcore*. This avoids the cost of repeated thread migration. If spare cache capacity is not available on the current core, but is available on another core (lines 12–16), MFC migrates to that core to cache f there.

In lines 18–29, if no core has f cached, but f is more frequently used than the least frequently-used file cached on any core (`lfu1`), `mfc_access` returns a request to migrate to the core c that holds `lfu1` and to use caching loads. This will cause f to be loaded into c's cache. `mfc_access` also marks `lfu1` as not cached on c, so it will no longer be used and the hardware will eventually evict it. If no core has f cached and f is infrequently used (lines 27–29), then `mfc_access` returns an indication that non-caching loads should be used, and, because it doesn't matter which core issues non-caching loads, that the current core *curcore* should be used.

If f is cached on some core, `mfc_access` looks up the core (*newcore*) that holds the file and is least loaded right now (line 31). MFC then decides whether it should make a copy of f on core *curcore* or migrate the thread to *newcore*. If *newcore* is busy, *curcore* is not busy (line 32), and *curcore* holds a file `lfu2` that is much less popular than f (line 35), MFC replicates f on *curcore* by not migrating away from *curcore*. `mfc_access` also removes *curcore* from the list of cores that cache `lfu2`.

Finally, on lines 42–43, if MFC believes f is cached on core *newcore* and has decided not to replicate it on core *curcore*, `mfc_access` returns *newcore* so that `read` will execute there.

4.3 Least-frequently used replacement

At any given time, MFC has chosen some set of files to reside in on-core caches; `read` accesses all other files using non-caching loads. MFC chooses to implement a least-frequently-used replacement strategy, although others are also possible. With LFU, the on-chip files are all more popular than the others, as measured by their `f.use[]` counts. Assuming that applications always read the entire contents of a file, this minimizes the number of cache misses to access file data, thereby achieving MFC's goal.

If a file f_1 that is not cached is used many times, the sum of its `f.use[]` counts may become larger than some files that are cached. At that point, MFC will start caching f_1. It will also mark the least popular file f_2 as not cached, so that it will be accessed with non-caching loads. Eventually the hardware cache replacement policy will evict f_2 from the on-chip cache.

To account for workload changes over time, `f.use[]` tracks an exponentially-weighted moving average of file accesses, by periodically decaying use counts in `update_use`. This ensures that MFC caches recently-popular files, as opposed to files that have been popular in the past.

```
1  // returns which core should execute read()
2  [core_id,load_type] mfc_access(file_t f, off_t o)
3    if f.size < min_read_size:
4      return [curcore, caching]
5
6    if curcore.cache_avail > f.size:
7      // curcore has spare capacity, replicate f
8      f.cores = f.cores + {curcore}
9      f.use[curcore]++
10     return [curcore, caching]
11
12   maxcore = core_with_max_cache_avail()
13   if maxcore.cache_avail > f.size:
14     f.cores = f.cores + {maxcore}
15     f.use[maxcore]++
16     return [maxcore, caching] // migrate to maxcore
17
18   if f.cores == {}:
19     // f is not yet cached anywhere by MFC
20     [lfu1, c] = lookup_lfu_file() // least popular file
21     if lfu1.use[c] < sum(f.use):
22       // c has a file lfu1 that is less popular than f
23       f.cores = f.cores + {c}
24       lfu1.cores = lfu1.cores - {c}
25       f.use[c]++
26       return [c, caching] // migrate to c
27     else
28       f.use[curcore]++
29       return [curcore, non-caching]
30
31   newcore = core_with_min_runqueue_len(f.cores)
32   if newcore.runqueue_len > busy_threshold &&
33      curcore.runqueue_len < idle_threshold:
34     lfu2 = lookup_lfu_file(curcore)
35     if f.use[newcore] > 2 * lfu2.use[curcore]:
36       // f is popular enough to evict local lfu2
37       f.cores = f.cores + {curcore} // replicate f
38       lfu2.cores = lfu2.cores - {curcore} // remove lfu2
39       f.use[curcore]++
40       return [curcore, caching]
41
42   f.use[newcore]++
43   return [newcore, caching] // migrate to newcore
44
45 update_use() // runs periodically to age files' use
46   for all f, i:
47     f.use[i] = f.use[i] / 2
```

Figure 2. Pseudo code for the `mfc_access` function. The function runs on the core `curcore` and considers migration to the core `newcore`. Each core runs the function `update_use` periodically.

4.4 Multiple copies

Because caches are associated with particular cores, MFC must be careful in evaluating the competing use of cores and cache units for computation and storage. To maximize capacity and reduce DRAM references, MFC wants to assign each file to only one core, but this may lower throughput. For example, a core holding a popular file may be a CPU bottleneck if many threads migrate to it.

MFC addresses such bottlenecks by replicating popular files in the caches of several cores so that they can jointly serve `read` calls for that file. It does this when the cores caching the file are busy (have long scheduler queues), and when there are other less busy cores that are caching significantly less popular files. MFC then takes advantage of these extra copies by migrating readers of a file to the least-busy core that caches the file.

4.5 Migration and avoiding its costs

MFC migrates a thread by moving it from the run queue of one core to that of another, where the kernel scheduler will find it. Except in extreme cases of imbalance, the kernel scheduler obeys this form of migration.

The cost of migration impacts some of MFC's decisions. For Linux 2.6 on the AMD platform we measured the time to move a thread between the run queues of two cores on the same chip as $9.2\mu s$, between two cores on different chips one HyperTransport hop away as $10.5\mu s$, and between two cores on different chips two hops away $11.7\mu s$.

`mfc_access` does not migrate a thread that reads a very small file in order to avoid migration overhead. MFC decides a file is too small if the time to load the file from off-chip memory would be less than the time to migrate and load the file from on-chip cache.

MFC estimates the costs of loading a file from on-chip and off-chip memory by dividing the file size by the on-chip and off-chip memory bandwidth, respectively. The memory bandwidth is measured off-line, although MFC could measure it during bootup.

MFC migrates a thread when loading from off-chip is more expensive than loading from on-chip plus the migration cost, including the time to move a thread between run queues, and the time to copy the thread's working set into local cache after migration. The latter cost is a property of the application's working set. MFC conservatively estimates this working set to be 4 Kbytes; §6 presents more detailed measurements.

An important special case is when a single-threaded application frequently reads a few popular files whose working set is larger than a single cache's capacity. In this case, MFC would like to use all core's caches to hold the most-frequently used files. But, if the files are used often, it may want to migrate a file to the thread's core instead of paying the cost of thread migration over and over.

MFC uses a simple competitive algorithm [14] to handle this case: once the cumulative cost of multiple thread migrations for a file exceeds the cost of loading it from off-chip memory, MFC stops migrating and loads the file from off-chip memory.

4.6 Sharing on-chip caches

The description of MFC has assumed so far that MFC can use the total on-chip cache capacity for caching files. In practice, MFC must share the caches with other uses such as program text and data. Furthermore, small files, which MFC ignores, also take up space. To handle sharing of on-chip caches, MFC must be able to distinguish a cache that is already full of active data from a cache that has a significant amount of inactive data that could be replaced.

During a scheduling time quantum MFC uses hardware event counters to count the number of cache misses in each level of a core's multilevel cache. MFC calculates a coarse-grained estimate of how much of level l cache a core c is using, $M_{\text{use}}(c,l)$, as:

$$M_{\text{use}}(c,l) = \begin{cases} M_{\text{size}}(l) & \text{if } M_{\text{miss}}(c,l) \geq m_{\text{lines}} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

For cache level l, this equation assumes a core is using the entire capacity, $M_{\text{size}}(l)$ bytes, of cache level l if the number of misses $M_{\text{miss}}(c,l)$ is more than the total number of cache lines m_{lines}. MFC then calculates the available on-chip cache capacity on a core c as:

$$M_{\text{avail}}(c) = M_{\text{on-chip}} - \max\left\{ \sum_{l}^{\text{all levels}} M_{\text{use}}(c,l),\ \sum_{f}^{\text{files on c}} S(f) \right\} \quad (2)$$

where $M_{\text{on-chip}}$ is the total amount of cache space accessible to a core (e.g., the sum of L1, L2, and L3 caches on the AMD system), and $S(f)$ is the size of file f. Subtracting the maximum of the estimate calculated using the event counters and the sum of all files on c from the on-chip capacity $M_{\text{on-chip}}$ avoids assigning too many files to c in the absence of cache activity.

MFC's estimate of available on-chip cache capacity is coarse-grained, but has three important properties. First, it indicates when the fastest levels of the cache hierarchy are full, which allows MFC to assign popular files to cores that have the lowest level of cache space (i.e., fastest) available. Second, it indicates when the caches are full, so MFC will stop assigning files to the core and avoid overflowing the total cache capacity. Third, it works well for shared caches, because the implementation takes into account that the sum of files on some core c may include files on other cores that overlap in the shared cache.

4.7 Large files

If a file is larger than the capacity of a core's cache, MFC splits the file into several smaller chunks and handles the chunks independently. The implementation's chunk size is half the size of a core's L2 cache. This design causes large files to be striped across caches. MFC keeps track of file chunks by treating each chunk as an independent file in its implementation of `mfc_access`.

4.8 File writes

In principle, MFC could migrate threads for file writes as well as reads, since both operations access file data in caches. However, handling writes turns out to be unimportant in practice. If file writes are not handled by MFC, they have the potential to invalidate cached file data on MFC-assigned cores, by modifying the same data on another core, or causing other file data to be evicted from that core's cache. However, performance of applications that involve a significant fraction of file writes is limited by disk performance, as opposed to either cache or DRAM speed. On the other hand, in applications with a small fraction of writes, invalidations of cached file data due to writes are infrequent, and MFC already allows for the divergence between its cache information and actual hardware state, due to unpredictable hardware evictions. In particular, the next file read might incur a needless migration to a core where the file data is no longer cached, but accessing the file would load the data into the correct cache again.

5. Implementation

We implemented our MFC design for Linux kernel version 2.6.29-rc7 and 64-bit AMD Family 10h CPUs, described in §2. MFC works on top of the existing Linux file cache. We modified the `read` system call to invoke `mfc_access` and perform migration if necessary. Applications that do not use the `read` system call (e.g., by using `mmap`) must still inform MFC when they are accessing files, and MFC provides an explicit system call that allows applications to invoke `mfc_access` directly in such cases.

Most of the MFC implementation (1,000 lines of C code), is in a file we added to the source tree. This file contains the `mfc_access` function and helper routines (600 lines of code), initialization functions (200 lines of code), and a `/proc` interface (200 lines of code). We added 50 lines of C code to the scheduling framework, to provide better thread migration and expose functionality for comparing CPU loads, `copy_to_user` functions that use `prefetchnta` (a

non-caching load instruction), and hooks to the file read code to call the appropriate the `prefetchnta copy_to_user` if the MFC designates a file as non-cacheable. The MFC implementation is stable and works with unmodified applications.

The `mfc_access` implementation was built with scalability in mind. When `mfc_access` does not replicate data or move data to a popular core the only state shared by multiple cores are per-replica popularity counters that are incremented each time the file is accessed. When `mfc_access` does replicate a file or move a file to a popular core, it acquires per-core spin locks, which protects lists from concurrent modification, for the cores involved and moves or replicates the file to a new list with approximately 50 instructions. The implementation is scalable in the sense that the cost of locking and accessing the per-replica shared counters is negligible compared to other overheads, such as migration.

MFC uses the CPU hardware event counters to count L1 misses, L2 misses, and L3 misses. MFC configures the event counters to ignore misses that are on modified cache lines. This potentially underestimates the working set size; however, it avoids overestimating the working set size when applications or the kernel shares data structures between cores that are modified often, such as spin locks.

6. Evaluation

This section explores the extent to which SCU helps software use multicore chips' cache space more effectively. The goal of SCU is to treat a set of private caches as a single shared cache, with resulting decrease in off-chip DRAM references. The desired end effect is higher application throughput. This section therefore compares the run-times of file-reading applications with and without MFC.

One would expect only certain kinds of applications to benefit from MFC. The ideal applications would be those that could benefit from a larger cache—that is, applications that spend most of their time reading files and comparatively little time computing; that read the same files multiple times, and thus can benefit from caching; and that have a total working set size larger than the size of each private cache, but not too much larger than the total on-chip cache. Moreover, because MFC uses thread migration to implement SCU, the ideal application would read relatively large files, one at a time, to amortize the cost of migration. These properties limit the range of applications that can benefit from MFC on current hardware, and determine the choice of benchmarks.

We evaluate MFC's performance in five steps. First, §6.2 measures the performance of a set of benchmark applications that have the above properties to varying degrees, with and without MFC. The benchmarks include file utilities such as `wc` and `grep`, as well as a microbenchmark that just reads files. MFC improves the performance of these single-threaded applications, because it allows the application to benefit from the unified cache resources of the chip, even though the application code runs on only one core at a time.

Second, §6.3 evaluates parallel versions of these benchmarks to show that (to a first approximation) the benefits of SCU apply even when all cores are actively computing. The parallel benchmarks also demonstrate that MFC load-balances cache space and cycles well.

Third, §6.4 shows that MFC is applicable to applications that access multiple files at the same time, by measuring the performance of a file reading microbenchmark that reads two files at once.

Fourth, §6.5 explores the performance of MFC on possible future hardware. One possibility we explore in §6.5.1 is that future multicore chips will have many more cores than the AMD machine used in the experiments, but will have memory systems that are not significantly faster. We hypothesize that MFC will provide benefits to a wider range of applications on such future hardware. In order to emulate future machines, we run the same benchmarks as before on the same AMD hardware but with only a subset of its 8 DRAM controllers enabled. The results show that the benefit of MFC increases significantly as DRAM bandwidth per core decreases, suggesting that SCU will become more important in the future.

The other possibility for future hardware that we explore in §6.5.2 is faster migration. We show that, if migration is cheaper (either using specialized hardware support, or using existing mechanisms like SMT/HyperThreading), MFC's benefit increases, which can make SCU applicable to a wider range of applications.

Finally, §6.6 examines individual MFC design decisions and characteristics in detail, such as the cost of updating MFC metadata and the importance of replicating files.

6.1 Experimental setup and applications

We run all of our experiments on the AMD system described in §2, using both a microbenchmark and a number of existing applications.

Our microbenchmark repeatedly selects a random file and reads it entirely with `pread`. The performance of the benchmark is measured as the number of megabytes read by `pread` per second. This microbenchmark reflects the workload that a busy multithreaded Web server might experience serving static Web pages. We will refer to this microbenchmark as the *read* microbenchmark.

We also use several unmodified file-intensive applications (`grep`, `wc`, and `md5sum`) to demonstrate that existing applications can achieve performance improvement with MFC. `grep` searches for a single character in the input files and `wc` counts new-line characters.

The benchmarks run the applications and read microbenchmark repeatedly on the same set of 256 Kbyte files and select files to process randomly. With smaller file sizes, the performance of benchmarks with MFC would decrease due to the overhead of thread migration in Linux. For files smaller than 45 Kbytes, MFC will stop migrating threads, and benchmark performance will be equal to performance without MFC. We explore migration costs in §6.5.2.

To simulate different application workloads, we configure our benchmarks to select random files according to either a uniform distribution or a Zipfian distribution[1], and report results for both configurations. Every data point represents the average of five runs.

6.2 Single thread performance

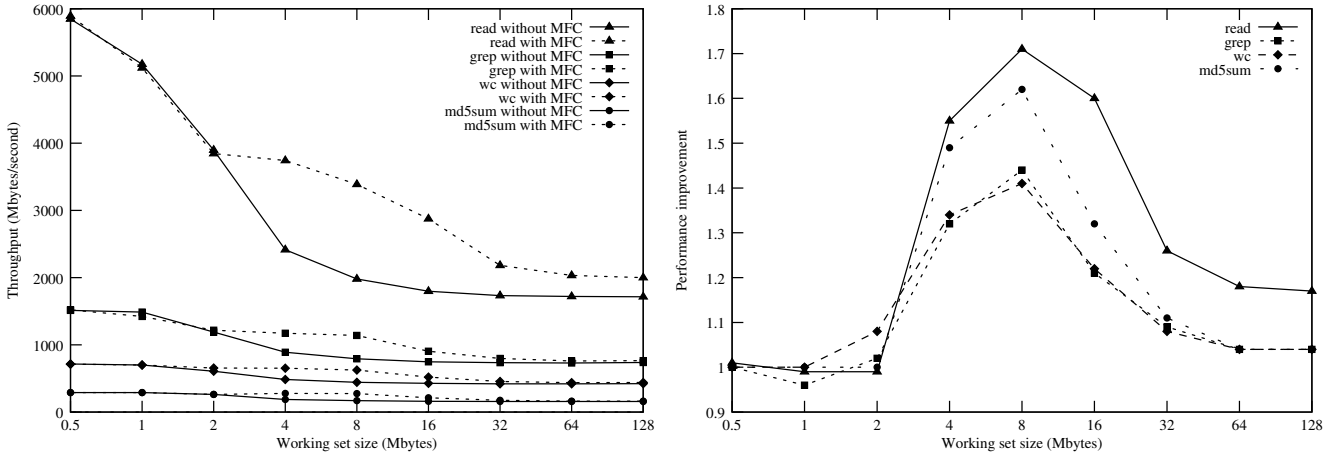
This section demonstrates that SCU can provide higher performance by helping software use cache space more effectively. We make this point using MFC with the read microbenchmark and three file-intensive applications. Figure 3 summarizes the results when the benchmarks select files according to a uniform distribution and Figure 4 summarizes the results when the benchmarks select files according to a Zipfian distribution. The number of files, and thus the total working set size, varies along the x-axis. The y-axis shows performance as the number of Megabytes each application reads per second. Figures 3(b) and 4(b) show the performance with MFC divided by the performance without.

At a high level, Figure 3 and 4 show that MFC reduces the runtime of the read microbenchmark relative to Linux without MFC for most working set sizes.

MFC offers no improvement for working set sizes of 2 Mbytes and less because each L3 cache can hold an entire copy of the working set. Improvement in Figure 3(b) starts to be greater than one at 4 Mbytes, because MFC can fit the entire working set in the aggregate on-chip cache. Without MFC each core can only use the fraction of cached data stored on its chip, and must wait for DRAM for the rest. MFC offers the greatest performance improvement for all four applications when the working set is 8 Mbytes.

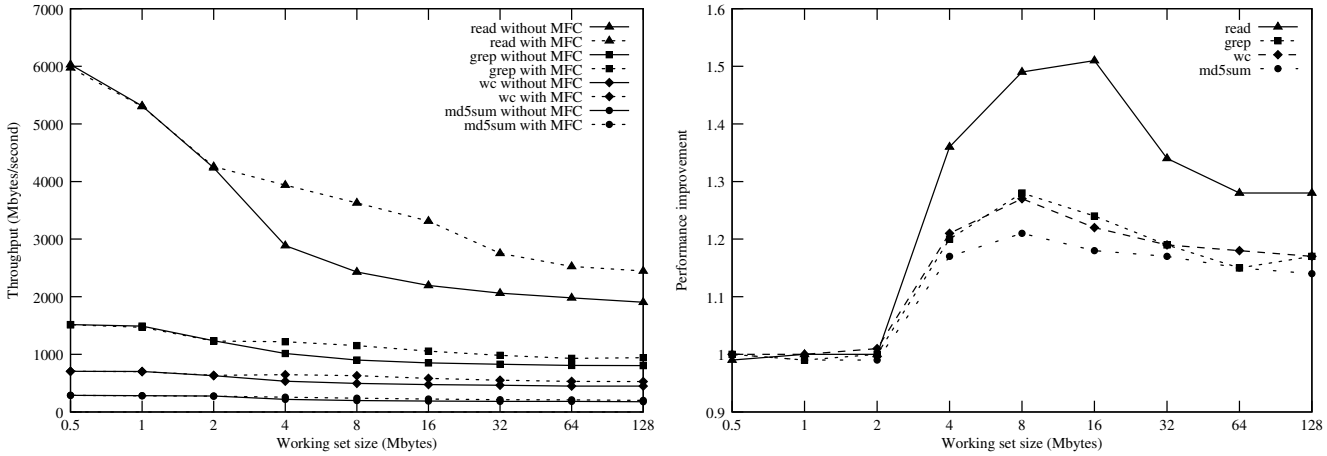
The amount of performance improvement with an 8 Mbyte working set depends on the amount of computation per byte read

[1] In a simple Zipfian distribution the nth most common element occurs $1/n$ times as often as the most frequent element.



(a) Absolute performance. (b) Performance improvement.

Figure 3. Uniform: Application and read microbenchmark performance with and without MFC. The working set size varies along the x-axis and the y-axis shows the read throughput. The benchmark repeatedly selects a file to read according to a uniform distribution.



(a) Absolute performance. (b) Performance improvement.

Figure 4. Zipfian: Application and read microbenchmark performance with and without MFC. The working set size varies along the x-axis and the y-axis shows the read throughput. The benchmark repeatedly selects a file to read according to a Zipfian distribution.

(see Figures 3(b) and 4(b)). The read microbenchmark does no computation and MFC increases its performance by 71%. MFC improves the performance of `md5sum` by 62%, `grep` by 44%, and `wc` by 41%. For working sets larger than 16 Mbytes improvement starts to decrease, because MFC must also wait for DRAM. MFC still wins for working sets much larger than total on-chip cache space, and never does worse.

These results suggest that MFC does a good job of keeping the most-frequently used files in on-chip memory, and delegating less-frequently used files to off-chip memory. For a given working set size, if running with MFC causes the benchmark to load fewer bytes from DRAM than without MFC, it can be concluded that MFC does a good job of caching frequently used data on-chip. This is confirmed by counting the number of bytes loaded from DRAM per byte of file data that the read microbenchmark reads, as shown in Figure 5. We used a hardware event counter on the AMD to count the number of DRAM loads. The memory controller on the AMD quad-core might prefetch data from DRAM, which is not counted by the hardware event counter.

Based on Figure 5, MFC achieves the most significant reduction in DRAM references relative to Linux with a working set size of 8 Mbytes. MFC causes the read benchmark to access DRAM about 190 times less often than Linux without MFC for uniform file selection, and about 115 times less often for Zipfian file selection. MFC cannot store working set sizes 16 Mbytes and larger on chip, and the number of DRAM access increases. For 128 Mbyte sized working sets, however, MFC still reduces DRAM loads by a factor of 1.2 for uniform file selection, and by a factor of 2.3 for Zipfian.

6.3 Parallel applications

To make sure that MFC can improve performance without wasting CPU cycles, we measure the performance of many copies of the same benchmark executing in parallel on 16 cores, and show the results in Figures 6 and 7. For Linux without MFC, 16 processes use all the available CPU cycles. For MFC, we need to run enough

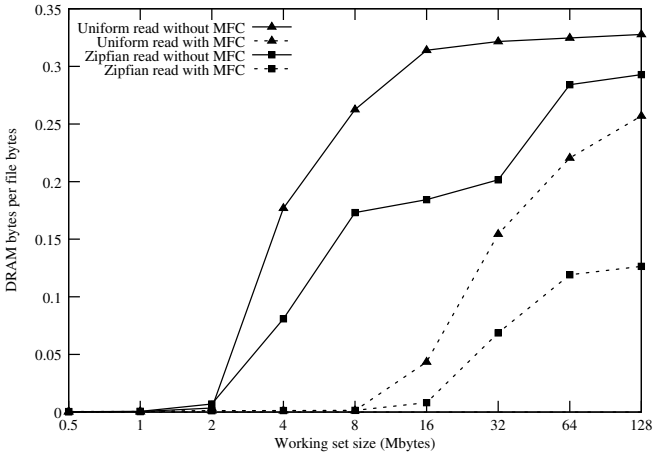


Figure 5. Aggregate bytes read from DRAM by all cores, per file byte that the read benchmark reads.

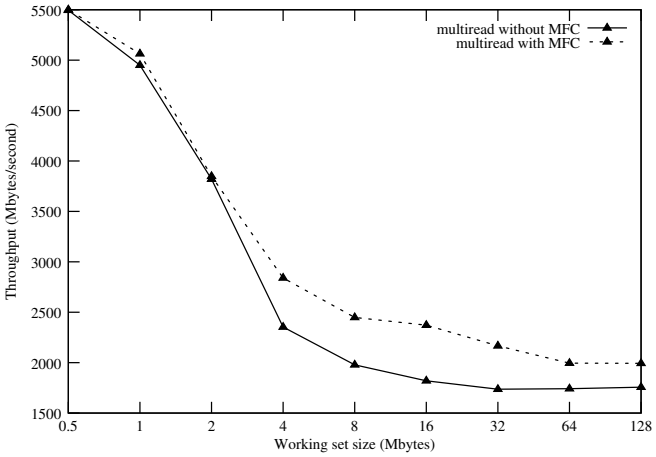


Figure 8. Performance of a single read microbenchmark process, using uniform file selection, reading two files but only migrating for the first file read.

processes to ensure no CPU is ever idle. In practice, 32 processes turns out to be sufficient, and fewer processes result in idle cores.

The trends for the parallel benchmarks are similar to the single-threaded ones. When running threads simultaneously on all 16 cores, each core has a smaller share of off-chip memory bandwidth and avoiding off-chip DRAM accesses becomes more important for good performance. For the read microbenchmark, MFC improves performance of the parallel version by up to a factor of 3.81, significantly more than the single-threaded version. This is because the aggregate read bandwidth required by the parallel version, shown in Figures 6 and 7, exceeds the aggregate 28,000 Mbytes/sec DRAM bandwidth of our machine.

Other parallel applications do not exceed DRAM bandwidth, but instead suffer a penalty, because Linux thread migration becomes less efficient with more processes. For uniform file selection MFC improves performance by a maximum of 15% for `grep`, 10% for `wc`, and 6% for `md5sum`. For Zipfian file selection MFC improves performance by a maximum of 14% for `grep`, 6% for `wc`, and 5% for `md5sum`.

6.4 Multiple files

SCU can improve performance when applications access multiple objects at the same time. We demonstrate this with MFC using a version of the read benchmark that reads two files at the same time, but only migrates to access the first file. To prevent the benchmark from migrating to access the second file we configure MFC (using a debug interface) to always ignore the second file read.

Figure 8 shows the results from this experiment when files are selected using a uniform distribution. MFC does not improve performance as much as for the single file case, but still provides a 30% improvement for working set sizes of 16 Mbytes.

6.5 Future hardware

MFC is targeted to future hardware platforms with more cores. We evaluate how MFC could perform on future hardware by emulating salient future properties.

6.5.1 DRAM bottleneck

We simulate DRAM bandwidth scarcity by throttling the memory system of the AMD machine. Applications are only starting to hit the overall DRAM bandwidth limits of current processors, so we expect to see only modest improvements for MFC with throttling. Each chip of the AMD machine has two DRAM controllers (a total of eight for the machine). We throttle the memory system by configuring Linux to use DRAM from one, two, or four chips, and either one or two controllers per chip.

Figure 9 shows the results of the parallel benchmarks with uniform file selection on a system using only one DRAM controller on a single chip, which is approximately $\frac{1}{8}$th of the original memory bandwidth. MFC improves the read benchmark's performance by a factor of 5.18, much more than in the non-throttled system. Similarly, `grep` improves by 97%, `wc` by 32%, and `md5sum` by 7%. The performance improvement of the `grep` benchmark on the throttled system is almost four times that of the non-throttled system.

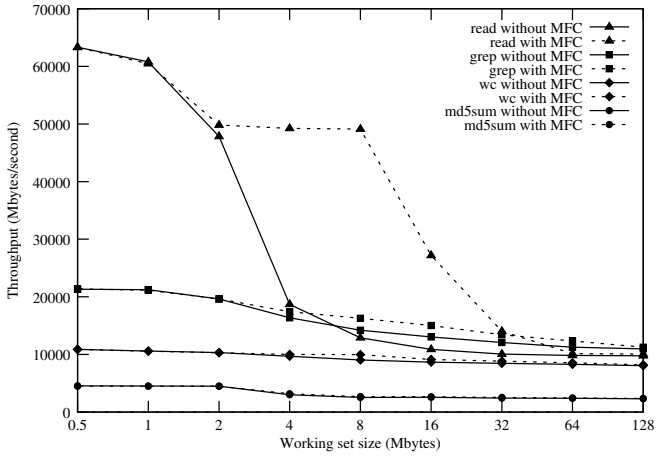
To explore how memory bandwidth impacts MFC, we measured the performance improvements of the same `grep` benchmark with the DRAM bandwidth constrained to 1, $\frac{1}{4}$, and $\frac{1}{8}$th of the original memory bandwidth. The `grep` benchmark achieves improvements of 15%, 37%, and 97%. These results indicate that SCU will provide more performance improvement in future processors, as the amount of DRAM bandwidth per core shrinks even further.

6.5.2 Faster migration

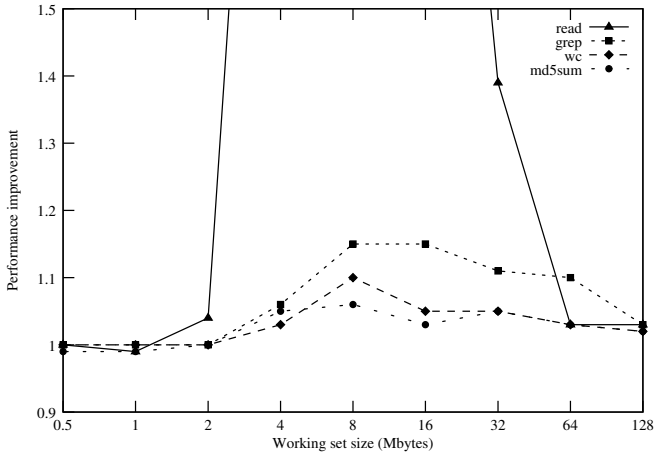
Linux thread migration was designed for periodic load balancing, and not for frequent migration such as MFC's. Migrating a thread in Linux takes about $10\mu s$ on the AMD system, which prohibits SCU from using migration to access fine grained data structures.

With faster thread migration SCU might improve the performance of more application workloads. One concern is that applications might have large working sets which will be expensive to copy from one core's cache to another regardless of a well optimized thread migration implementation. We explored this concern by measuring the number of bytes copied between cores for the read benchmark, `grep`, `wc`, and `md5sum`. We modified the read benchmark, `grep`, `wc`, and `md5sum` to process a single 64 Kbyte file continuously. Before one of the benchmarks begins to process the file, it migrates to another core. We used a hardware event counter to count the number of cache lines copied between the source and destination while the file was processed.

Figure 10 presents the average number of bytes the destination core copies from the source core per migration for each application. At a high level, the results give a lower bound on file size MFC could manage if thread migration in Linux had no cost. For example, migrating `md5sum` requires copying 1344 bytes of `md5sum`'s working set, so it will always be cheaper to copy files less than 1344 bytes between cores, instead of migrating. The results indicate

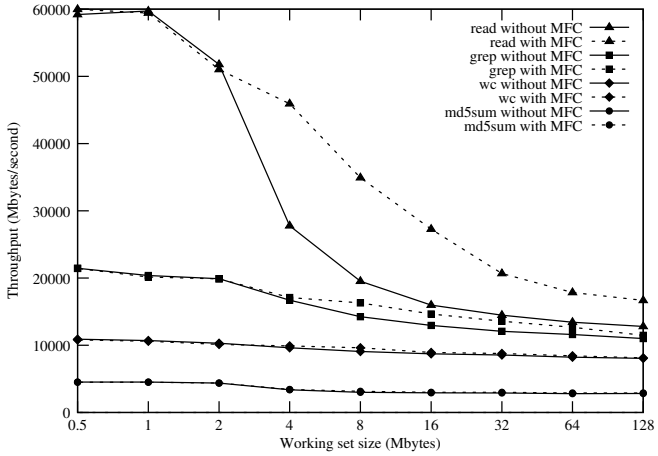


(a) Absolute performance.

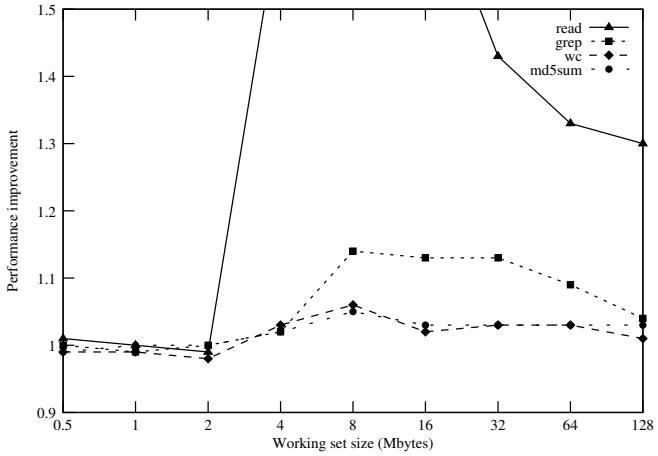


(b) Performance improvement. The improvement for the read microbenchmark is 2.63× for 4 Mbytes, 3.81× for 8 Mbytes, and 2.50× for 16 Mbytes.

Figure 6. Uniform: Parallel application and read microbenchmark performance with and without MFC. The working set size varies along the x-axis and the y-axis shows the read throughput. The benchmark repeatedly selects a file to read according to a uniform distribution. We ran 16 instances of each application on Linux without MFC and 32 instances of each application on Linux with MFC.



(a) Absolute performance.



(b) Performance improvement. The improvement for the read microbenchmark is 1.65× for 4 Mbytes, 1.79× for 8 Mbytes, and 1.71× for 16 Mbytes.

Figure 7. Zipfian: Parallel application and read microbenchmark performance with and without MFC. The working set size varies along the x-axis and the y-axis shows the read throughput. The benchmark repeatedly selects a file to read according to a Zipfian distribution. We ran 16 instances of each application on Linux without MFC and 32 instances of each application on Linux with MFC.

Application	read	grep	wc	md5sum
Bytes copied per migration	256	832	594	1344

Figure 10. User-level cost of migration measured in the number of bytes copied per migration.

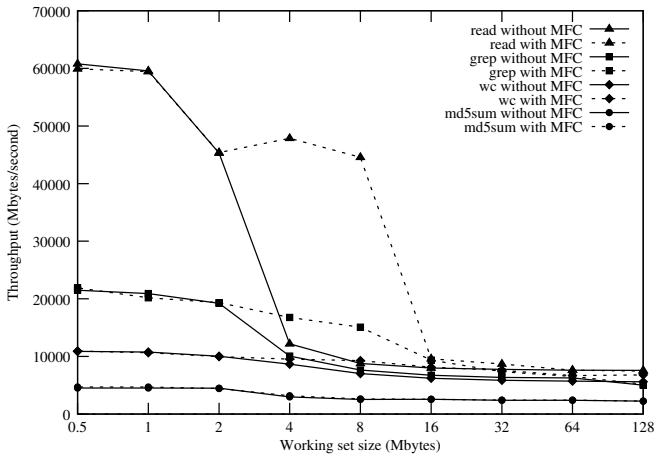
that more fine grained data management is possible given more a more efficient thread migration implementation.

6.6 MFC design

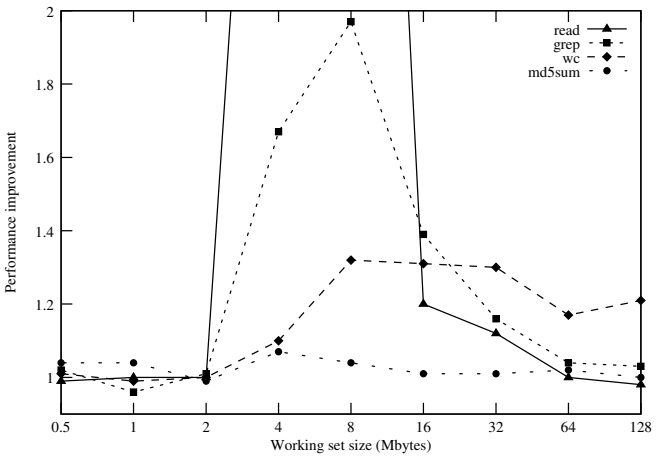
This subsection evaluates a few important MFC design properties individually.

6.6.1 Metadata overhead

MFC maintains metadata for each file and core. MFC updates metadata each time an application accesses a file and when MFC changes which cores a file is assigned to. Figure 11 presents the average time MFC spends updating metadata. We generated load on MFC with 32 instances of the read benchmark and Zipfian file selection. The file working set was 128 files of 256 Kbytes each. The $0.495\mu s$ overhead for updating metadata is small compared to the costs of reading a 256 Kbyte file from on-chip memory or DRAM, which are $45\mu s$ and $145\mu s$ respectively. It is more costly for MFC to reassign a file, about $4\mu s$, but this happens only when file popularity changes. On our 16-core AMD machine, MFC's metadata amounts to 234 bytes per file, dominated by per-core access and cache data.



(a) Absolute performance.



(b) Performance improvement. The improvement for the read microbenchmark is 4.08× for 4 Mbytes and 5.27× for 8 Mbytes.

Figure 9. Benchmark performance with a throttled memory system. File selection is uniform. We ran 16 instances of each application on Linux without MFC and 32 instances of each application on Linux with MFC.

Operation	Execution time
File access	$0.495\mu s$
File assignment	$4.101\mu s$

Figure 11. The average time taken by MFC to update file and core metadata when running 32 instances of the read benchmark with Zipfian file selection. The read benchmark reads each 256 Kbyte file in $45\mu s$ if the file is in on-chip memory, and in $145\mu s$ if the file is not.

6.6.2 Multiple copies

To demonstrate that MFC's procedure for replicating file contents is important, we perform an experiment where MFC disables popular file replication, and compare the results to regular MFC. The experiment runs 32 processes of the read microbenchmark with Zipfian file selection. With replication disabled, each file has a dedicated core, and threads always migrate to that core to read it.

Figure 12 shows the results from the experiment. The "without replication" line gives the performance of when MFC disable popular file replication. For comparison the figure includes the performance when MFC is configured normally and the performance without MFC, from previous Figure 7(a). For small working set sizes MFC without replication performs poorly because the working set is composed of only a few 256 Kbyte files, so the few cores that hold the most popular files are bottlenecks. The amount of concurrency (and improvement) increases as the working set size increases because the working set is composed of more files.

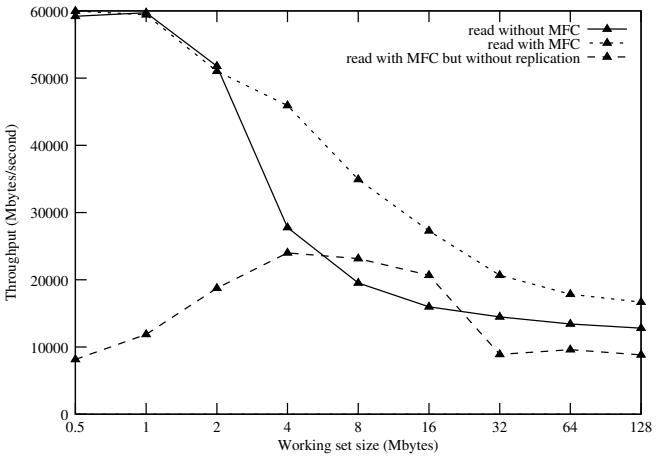


Figure 12. Performance of the 32 processes running the read microbenchmark with Zipfian file selection, when MFC disables file replication. Performance of MFC with replication enabled (default), and of Linux without MFC are shown for reference.

7. Discussion

While we expect that the SCU approach could be used for a wide range of applications, the current MFC prototype is limited to a narrow range of read-heavy file system applications. This is largely motivated by the convenience of a file system API, which makes data access explicit, and the limitations of commodity hardware that MFC runs on. The rest of this section discusses the applicability of SCU at other levels in the system, and how future hardware support could improve SCU.

To control the placement of data in a distributed cache, SCU must migrate threads before they access each data object. The file system API provides a particularly convenient point of migration, in the `read` system call. We expect that other systems with a well-defined notion of an object could similarly benefit from SCU, and some work has been done in this direction [4]. For example, a language runtime, such as the Java VM or a Javascript engine, could migrate threads at calls to methods that access large objects. Database servers could also migrate threads when scanning different tables or indices. While optimizing the cache utilization of write-heavy applications did not make sense in the file system, applications such as an in-memory database might achieve higher write performance with SCU as well. For applications that do not have a natural object access boundary, current hardware does not provide convenient hooks for intercepting arbitrary cache misses. However, with the right hardware support, SCU might be able to support these applications as well.

In addition to intercepting cache misses, two other factors limit SCU on the hardware side. First, today's multi-core processors have relatively high costs for migrating computation between cores. Thus, applications must not migrate too often, if they are to recoup

the cost of migration. Second, processors provide little software control or introspection for the caching hardware. This requires maintaining a separate data structure to represent cache contents, which is both costly and inaccurate, because hardware is making its own independent decisions about eviction and prefetching.

We expect that both of these limitations could be addressed in future hardware, which would allow a wider range of applications to take advantage of SCU. For example, support for efficient thread migration would make it worthwhile for SCU to migrate a thread for accessing a smaller amount of data on a remote core. This would make SCU applicable to applications that do not access data in large coarse-grained chunks. One way to make thread migration cheaper that we are exploring is to leverage hardware multi-threading (SMT) support.

Better support for micro-managing hardware caches would likewise improve SCU. If hardware allowed SCU to query the contents of a given cache, or to find out which cache currently contains a particular data item, SCU could find the migration target for a given thread with less overhead and with higher accuracy. SCU would also benefit from being able to explicitly push data to a certain cache, pin data in cache, or evict data, in order to implement a globally-optimal cache replacement policy.

With such hardware support, SCU might be able to obtain the benefits of globally-shared caches without the associated hardware complexity. Building globally-shared caches in hardware involves trade-offs, as large far-away caches can be slow to access, or requires complex designs to allow many concurrent accesses. Even processor that implement such expensive shared caches in hardware cannot migrate computation closer to the data, as SCU is able to. In future work, we plan to explore better hardware support for SCU, and how it affects the applicability of this technique.

8. Related work

The closest related work is the user-level O^2 runtime [4]. With O^2 a programmer must mark the beginning and end of an operation on an object, and then the O^2 runtime schedules objects and operations together. If an object is large enough, O^2 migrates the operation to the core that holds the object. This work helped us in proposing software cache unification as a general idea, which can be applied at multiple levels of abstractions (files, language objects, and cache lines).

Although O^2 and MFC have a similar goal—making more effective use of multicore caches—the systems are different: MFC is a file cache extension for the Linux kernel, while O^2 is a user-level runtime. As a result, their designs are different and solve different problems. To the best of our knowledge MFC is the first file cache design to take advantage of the many on-chip caches on multicore processors.

MFC is related to techniques to optimize cache use on multicore chips, MFC's use of migration is similar to computation migration in software distributed shared memory, and MFC uses implementation techniques from other systems. We discuss each relationship in turn.

8.1 Multicore cache management

Several techniques have been proposed to achieve better cache behavior on multicore processors.

Thread clustering [22] dynamically clusters threads with their data on to a core and its associated cache. Chen et al. [8] investigate two schedulers that attempt to schedule threads that share a working set on the same core so that they share the core's cache and reduce DRAM references. These techniques do not migrate computation, but MFC might benefit from a similar approach for applications that share a file.

Several researchers have been looking at operating systems techniques to partition on-chip caches between simultaneous executing applications [9, 13, 18, 21]. Zhang et al. propose an efficient page-coloring and practical approach to supporting such cache management [26]. This line of work is orthogonal to SCU and MFC, but could be used to make shared caches behave like private caches, and give MFC more precision in assigning files to caches.

Jaleel et al. propose a thread-aware dynamic insertion policy (TADIP) for managing shared on-chip caches on a multicore processor [12]. TAPID's goals are similar to MFC's: understand how on-chip caches are being used, try to cache data that improves performance, and not cache other data. TAPID, however, is a hardware cache-management scheme and tries to decide based on a core's instruction stream whether to use least-recently used or a bimodal insertion policy.

Chakraborty et al. [6] propose computation spreading, which uses hardware-based migration to execute chunks of code from different threads on the same core with the aim of reducing instruction cache misses and branch mispredictions; the authors used computation spreading to place repeated user-level code on the same core and repeated system-call code on the same core. MFC doesn't try to improve instruction-cache performance, but one could imagine extending MFC's heuristics to take instruction caches into account.

Several researchers have proposed to assign OS functions to particular cores and have other cores invoke those functions by sending a message to that core. The Corey [3] multicore operating can dedicate a core to handling a particular network device and its associated data structures. Mogul et al. proposes to optimize some cores for energy-efficient execution of OS code and put the OS code on those optimized cores [15]. Suleman proposes to put critical sections on fast cores [20]. MFC can be viewed as dynamically assigning data to cores to get better cache performance.

Barrelfish attempts to solve a much bigger scheduling problem than MFC [16]. Its goal is to dynamically schedule applications across heterogeneous systems of multicore processors, taking into account diversity in terms of access time, cores, and systems. The authors of Barrelfish have noted the possibility of improved on-chip cache sharing on multicore processors through on-line monitoring, but haven't proposed a scheme for doing so.

The fos operating system runs like a distributed system on a multicore chip, treating each core and its cache as an independent computing node that communicate with other nodes using message passing [24]. Each core runs a small microkernel with the rest of the operating system implemented as distributed services. A benefit of this design is that it doesn't need globally shared memory and reduces cache interference between operating system and applications. SCU can be viewed an approach to obtain scalably the benefits of private caches while maintaining a shared-memory interface.

8.2 Computation Migration

MFC migrates a thread to the target core to read the content of that core's cache. This technique is similar to computation migration in distributed shared memory systems such as MCRL [11] and Olden [5]. These systems can migrate computation to a machine that has the data in its local off-chip memory in order to reduce total communication traffic. In the Olden system the decision to use computation migration is made by the compiler statically and the language was tailored to make it possible for the compiler to make static decisions. In the MCRL system, the decision is made dynamically, using a simple policy which maintains a limited history of accesses and sometimes migrates computations for read operations, and always for write operations.

In object-based parallel programming languages, the use of computation migration comes naturally, since the runtime system can invoke methods on remote processors/computers. A number of systems have used this ability to decide dynamically between

fetching an object's data and invoking a method on the local copy versus sending a message for a remote method invocation (*e.g.*, [2]). Some of these systems have combined this with memory-aware scheduling for NUMA machines (*e.g.*, [7, 10]).

Although MFC shares the technique of migration computation with these systems, MFC uses it to overcome the processor-memory performance gap, employs it in the context of an operating system's file cache, and uses different decision procedures for when to migrate.

8.3 Implementation techniques

Strong et al. have improved the performance of thread switching between cores in the Linux kernels by a factor of 2 [19]. Adopting these techniques would improve the MFC results in §6 further, and we plan to adopt the techniques. Tam et al. propose a light-weight scheme for estimating L2 miss rates [23]. MFC might be able to benefit from such a scheme to obtain a more precise estimate of available cache space than the one in Section 4.6, yet be efficient.

9. Conclusion

This paper presents the SCU approach to unify the distributed caches on multicore chips. As a concrete instance of SCU, the paper introduces MFC, which caches the most frequently-used files in distinct core's caches to maximize the use of on-chip memory, avoiding references to off-chip memory. MFC does so by migrating a thread to the core that holds the desired file, avoiding unnecessary replication of the file in multiple caches and consequent eviction of other useful data.

An implementation of MFC in the Linux kernel can speed up file-intensive applications by up to a factor of 1.6 on a platform with four AMD quadcore processors. Microbenchmarks show that MFC performs well for a wide range of working set sizes relative to the total on-chip cache, because MFC reduces the number of DRAM loads. Finally, on future machines with less DRAM bandwidth per core, MFC improves performance even more. These results suggest that on future multicore processors SCU will be an important approach for handling the increasing processor-memory gap.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant number 0915164.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, pages 92–103, 1996.

[2] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Trans. Comput. Syst.*, 16(1):1–40, 1998.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[4] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verità, Switzerland, May 2009.

[5] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[6] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize cmp cores on-the-fly. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 283–292, 2006.

[7] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: An object-based language for parallel programming. *Computer*, 27(8):13–26, 1994.

[8] 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 *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, 2007.

[9] S. Cho and L. Jin. Managing distributed, shared L2 caches through os-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[10] R. J. Fowler and L. I. Kontothanassis. Improving processor and cache locality in fine-grain parallel computations using object-affinity scheduling and continuation passing. Technical Report TR411, University of Rochester, 1992.

[11] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in DSM systems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 1996.

[12] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. PACT'08*, Oct. 2008.

[13] 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 *International Symposium on High-Performance Computer Architecture*, pages 367–378, February 2008.

[14] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms for on-line problems. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 322–333, New York, NY, USA, 1988.

[15] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-isa cmps to save energy on operating systems. *IEEE Micro*, pages 26–41, May-June 2008.

[16] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

[17] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[18] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with OS-level, software-only pollute buffer. In *41rst International Symposium on Microarchitecture (MICRO)*, pages 258–269, November 2008.

[19] R. Strong, J. Mudigonda, J. Mogul, N. Binkert, and D. Tullsen. Fast switching of threads between cores. *ACM SIGOPS Operating Systems Review*, 32(2), April 2009.

[20] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 253–264, 2009.

[21] D. Tam, R. Amzi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on Interaction between operating systems and computer architecture*, pages 27–23, June 2007.

[22] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–58, New York, NY, USA, 2007.

[23] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *ASPLOS '09: Proceeding of the 14th international*

conference on Architectural support for programming languages and operating systems, pages 121–132, 2009.

[24] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[25] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[26] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.