Predictable Coherent Caching with Incoherent Caches [*]

Adam Lackorzynski[†], **Benjamin Engel**[†] and **Marcus Völp**[◇,†]

[†]Technische Universität Dresden
Operating-Systems Group
Nöthnitzer Strasse 46, Dresden, Germany
{adam, engel, voelp}@os.inf.tu-dresden.de

[◇]Carnegie Mellon University
Logical Systems Lab
5000 Forbes Ave, 15213 Pittsburgh, PA, USA
mvoelp@cs.cmu.edu

Abstract

Caches are a well known hardware construct for improving energy consumption and average performance by keeping frequently-used data near processing resources. Yet, at the same time, they form a major hurdle for worst-case execution time analyses, in particular if they are shared between multiple cores. Exploiting that most shared data objects are accessed or at least committed in a mutually exclusive manner, we demonstrate, using an ARM MP-Core and an x86 multicore system, an explicit write-back synchronization scheme for shared data. We will analyze the performance and overheads it incurs, stressing predictability aspects. To our surprise, neither the local nor the shared caches need to be kept coherent for this scheme to work.

1 Introduction

Over the past years, significant progress has been made in predicting uniprocessor caches [1, 2] and in turn the worst-case execution times of applications. However, with the transition of Moore's Law from frequency growth to increasing parallelism, uniprocessor systems are slowly vanishing, even in hard real-time systems, and single core timing analyses cease to produce reliable results.

One major source of unpredictability in multicore systems is application interference over shared caches. As long as only one application accesses memory, allocation and replacement in the instruction and data caches follow this application's execution pattern and are therefore predictable through an offline static analysis. However, when caches are shared, multiple applications compete for cachelines and may possibly evict data that applications on other cores still need. Even worse, to manage coherence traffic efficiently, most multicore architectures keep track of where private copies are stored in the core-local caches. While maintaining this information helps avoiding unnecessary cacheline invalidation, exhaustion of the structures required to maintain this information leads to unpredictable invalidation of cached data.

Rather than abandoning shared caches for predictable multicore systems (as for example suggested by Cullmann et al. [5]), we believe it is time to rethink large parts of the memory and cache architecture to see whether current and future usage patterns allow for different solutions. This paper, is a first step in this direction. Ignoring for the time being that mechanisms such as coherence protocols are required to realize state-of-the-art synchronization protocols, we focus in this work on determining essential functionality if we assume that all accesses to shared objects are mediated by such a protocol. More concretely, we shall look into possibilities for consistently sharing objects that are protected by mutual-exclusion locks or alternatively by reader-writer locks. We shall also see how such a sharing can be established in transactional schemes where updates remain invisible until a point in time where they can be committed atomically.

[*]This work is in part funded by the DFG through the collaborative research center Highly Adaptive Energy Efficient Systems (HAEC) and through the cluster of excellence Center for Advancing Electronics Dresden (cfaed) and by the state Saxony and the European Union through the ESF young researcher group IMData.

We found that an unusual combination of cache coloring for shared objects plus selective write-back and invalidation of shared data from private caches leads to predictable caching of exclusive and shared data. Moreover, we found that traditional cache coherence is only required at the level of shared caches and not at all if modified shared data is consequently written trough to the most common shared cache or memory level, which unfortunately in most systems is main memory.

2 Caches and their Analysis

Although caches are widely understood, there are still some ambiguities in the used terminology. Let us therefore quickly recapitulate the role of caches in modern memory hierarchies and introduce the terminology we shall use in the remainder of this paper. We shall also introduce a cache analysis by Ferdinand et al. [1] in greater detail, which we extend to work with a selective invalidation scheme similar to the one proposed by Cheong et al. [3].

2.1 Caches

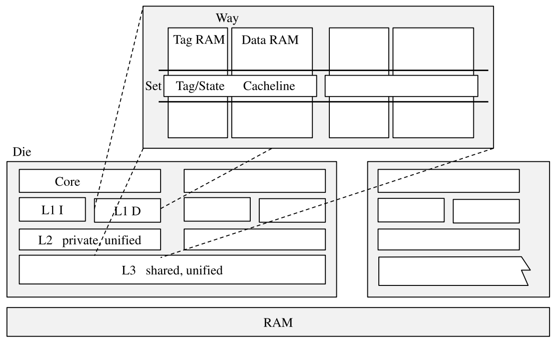

FIGURE 1: *Cache hierarchy and internal structure of typical memory architecture.*

Figure 1 depicts the cache hierarchy of a typical multicore system. The L1 instruction and data caches next to each core are *private*. That is, allocation of cachelines is triggered only by the applications on this core. In the figure, the L2 caches are still private, but *unified* (i.e., used for both data and instructions). For simplicity, we focus here on data accesses only. Incorporating instruction fetches should be straight forward provided the relative ordering of instruction fetches and data accesses is predictable [5]. The L3 caches are *shared* by all cores on the same die.

Data in memory is cached in chunks of size 2^s called *cachelines*. The chunk's memory address a is aligned to this size (i.e., $a == 0 \ mod \ 2^s$). In an n-way set associative cache, a uniquely identifies a *set* of n cachelines, which are subject to replacement. That is, if a cacheline is to be fetched from memory and allocated in the cache, it will be placed in this set, possibly replacing one of the other cachelines in this set but not cachelines allocated to other sets. We say that addresses, which are mapped to the same set share the same *cache color*. Later on, we shall exploit that cachelines of different color cannot evict each other [6]. Replacement within a set is subject to a *replacement policy* such as least recently used (LRU) or first-in-first-out.

Let us first focus on data that is exclusively used by one application. There are two situations in which the use of shared caches by concurrently executing applications may jeopardize predictability of access times to this exclusive data:

1. by evicting lines from the shared cache that the application expects to be cached; and
2. by evicting lines from the private caches to maintain the knowledge required for scalable coherency.

In the first situation, accesses by concurrent applications may cause the allocation and thereby eviction of cachelines used by other cores. Therefore, if an application on one of these cores accesses a cacheline, it may experience an unpredictable miss and possibly also a write back of dirty data to make room for the required allocation. A cacheline is said to be *dirty* if it has been modified. The writeback is necessary to ensure that the modification will eventually become visible to prospective readers. During these writebacks (either from a private cache or from a shared cache to the next lower shared cache) the accessing core may experience a further miss penalty if the written back cacheline is not cached at this destination. However, if the writeback does not affect one of the exclusive cachelines of the same core, it may also experience an unexpected hit causing this cacheline to be the most recently used. Depending on the replacement policy, interfering with this age information may not only render cacheline eviction unpredictable, it may also result in the replacement of a cacheline still used by one of the cores.

The second situation in which a concurrent access jeopardizes the predictability of caches is also triggered by evicting another core's cacheline. In order to keep caches *coherent*, so called *cache coherence protocols* ensure that prior to writing all cached copies are invalidated. To scale up this invalidation procedure, modern cache controllers maintain information about which higher level caches contain copies. The problem here is that this information is not commonly required. In fact, it is necessary only for shared data and therefore too expensive to maintain for all cachelines. For this reasons, the structures to maintain this information are typically laid out to capture only a fraction of all possible data that could be cached in private caches. However, this under-dimensioning leads to cacheline invalidation if their location information can no longer be kept. The structure for maintaining this information is called a *snoop filter*. Essentially, a snoop filter is a replication of the tag RAM of all caches it controls plus storage to hold information about whether or not a cacheline in a private cache and if so in which one. To avoid this duplication of the tag RAM, snoop filters are sometimes integrated into a shared cache, which is then said to be *inclusive*. An inclusive cache holds at least all those cachelines for which copies exist in the higher level caches it subsumes. To maintain inclusiveness in case of eviction, copies have to be invalidated, even from the core-local private caches.

Shared data accesses experience the same kind of eviction unpredictability. However, the effects are typically less severe because subsequent writes by other cores invalidate and thereby evict shared data anyway. Our goal is to prevent shared data accesses from interfering with private accesses and to exploit properties of synchronization patterns to preserve consistency and, to a certain degree, predictability for shared data accesses.

2.2 Cache-Aware WCET Analysis

We now turn our attention to a cache-aware worst-case execution time analysis for caches following the LRU replacement policy. See Ferdinand et al. [1] for a discussion of other replacement policies.

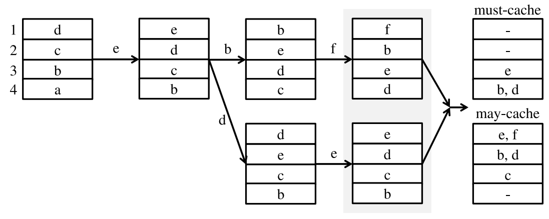

FIGURE 2: *May and must-cache analysis for private caches.*

Figure 2 shows the may/must-cache-analysis by Ferdinand and Wilhelm [1] for a short code path, which exhibits the access pattern $abcde(bf|de)$ for one specific color. Assuming that filling of invalid cachelines is prioritized over occupied lines (i.e., they are strictly older than filled lines), accessing a–d leads to a configuration of the cache where a is the oldest (next to be replaced) and d the youngest cacheline. Accessing e causes a cache miss and the eviction of a. If a has been modified, this triggers a write back to the next lower cacheline and hence a request which serves as input to the same kind of analysis for the next lower level.

The second box in Figure 2 denotes the state after e has been accessed. Non-trivial programs typically diverge in their control flow depending on external or otherwise unpredictable situations. In our example, the program executes bf in one branch of an if-statement whereas in the other it accesses d and e. The two middle columns in Figure 2 denote the respective cache state for these variants yielding the two grey-shaded cache configurations at the join point after the if-statement. If we can predict with certainty the condition and hence the branch that is taken, the analysis could fold the conditional execution into a linear one, discarding the non-taken branch. However, without knowing all possible inputs to the program we cannot make this decision. Therefore, we have to continue the analysis from the join point with two possible configurations. Whereas for small programs an evaluation of all possible execution paths may still be feasible, the scalability of such an approach to larger programs is quite limited. For this reason, Ferdinand and Wilhelm introduce a join criterion to combine the state of conditional branches into a must-cache and a may-cache state [1]. The must-cache state represents those cachelines that are guaranteed to remain in the cache. Accessing these cachelines will result in a cache hit and a juvennescence of these lines. It is computed by taking the maximum age of the cachelines in both of the branch states (grey). Cachelines that are not present are discarded from this state. The may-cache analysis denotes which cachelines may still be in the cache and hence which write-backs may be deferred and cause overheads at a later point in the execution. It is computed by introducing the cachelines at their minimum age.

Based on the cache state, it is now possible to compute the worst-case execution times of memory accesses by assuming a hit if the must-cache state contains the accessed cacheline, a miss if the may-cache state does not contain this cacheline and by taking the maximum execution time of hitting and missing in the other cases. Write-backs work the other way around. A write back is certain if the

[1]Notice that a program with nesting level of n yields only these two states throughout its analysis.

cacheline leaves the may-cache state and it is guaranteed not to happen yet if the cacheline remains in the must-cache state. If neither condition holds, we have to assume a potential write back.

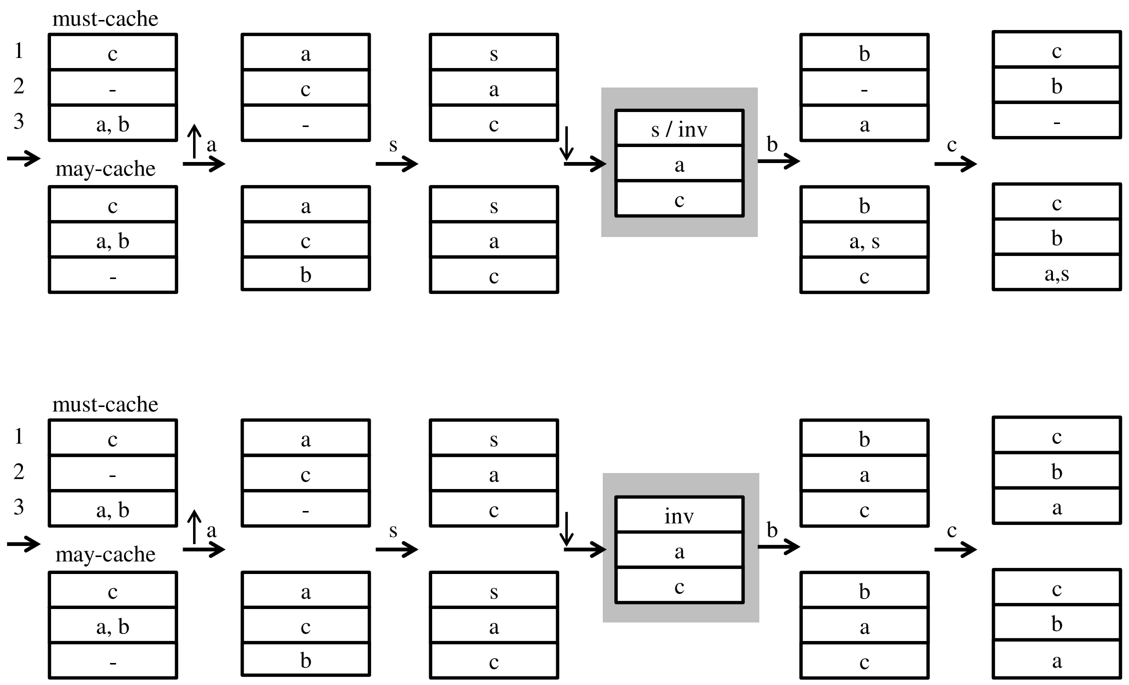

FIGURE 3: *May and must-cache analysis for private L1 caches without (top) and with (bottom) selective invalidation of shared objects.*

It is easy to see how external invalidation jeopardize the analysis. At any point in time relative to the execution of the analyzed program, the above described situations may occur and cachelines may be evicted or invalidated. Assuming mutually exclusive accesses or commits of shared objects, we regain part of this predictability by selectively invalidating shared state.

3 Rethinking Caches

In the presence of these difficulties, it is reasonable to question the presence of caches in the first place. Let us not do so in this paper. Caches achieve a much better average case performance than bypassing directly into uncached memory. Moreover, caches tightly integrate with translation lookaside buffers (TLBs) to offer cacheline granular access to objects that are subject to page granular access control, a combination not easily reproduced with tightly coupled on-chip memories[2]. How should caches look like to be more predictable for real-time systems?

3.1 Private Caches

To keep shared objects coherent, we have to ensure that reads always return the last value written. If we assume for the time being that objects are only always accessed while holding a protecting lock and if we further assume that the implementation of this lock is supported by other means, writes may remain completely local (i.e., invisible from the perspective of other cores) until another core is able to acquire the lock. At this point in time, all modifications must be visible to at least those cores that have a chance of obtaining exclusive access next.

With coherent caches, one option would be to leave the accessed parts in the last lockholder's cache from which the coherence protocol would snoop them by triggering invalidates and possibly also write-backs once the current lockholder accesses the data. However, this again causes interference with age information because invalidates can happen at arbitrary times relative to the execution of a core. Another option would be to trigger these invalidates and possible write-backs eagerly to ensure data is out be-

[2]We do address this question in the context of the ESF young researcher group IMData through DMA-like data transfer units and explicitly controllable channels [7] but the results do not yet justify abandoning caches.

fore another core can obtain access. For example, if a critical section involves accessing *sabt* where s and t are cachelines of a shared object and a, b are exclusive data accesses, then prior to releasing the lock we invalidate and if necessary write back s and t. Now, if every core releases shared data from all its private cores by writing modifications back to a lower cache level, the necessity of tracking locality information vanishes. We can even give up on maintaining inclusiveness in shared caches and, as we shall see, even coherence as it is enforced by coherence protocols.

Let us exemplify the consequences of our approach on Ferdinand's may/must cache analysis for private L1 caches. Let $\uparrow$ and $\downarrow$ denote the acquiring and release of a lock protecting the shared object s. Figure 3 shows the may and must analysis for a private L1 cache with and without selective invalidation. In both situations, the analysis considers that s is locked. We observe the sequence $abc(ac|bc) \uparrow as \downarrow bc$ starting after the join.

The analysis for the prefix $abc(ac|bc) \uparrow a$ works as described in Section 2.2. The next access is to the lock protected object s, which at this time is the most recent access. Had we not already considered that s is locked, we would have to assume that subsequent accesses to s miss. The additional knowledge that s is locked allows us to consider it for the must analysis. After releasing the lock however, s may either still be cached or already invalidated due to another core writing to it. We do not consider invalidation to maintain inclusiveness in this example. For the must analysis without selective invalidation, we can therefore not assume that s is still present. We may however also not assume that it is gone. Therefore, when accessing b, a drops to age 3 and c gets evicted. We can no longer conclude that the subsequent access to c hits in the cache. For the may analysis, s may still be there and drops to age 2 after accessing b. However, because s may also already have been invalidated, a and c maintain their age. If on the other hand we invalidate s before releasing the lock, we can be sure that s no longer resides in the cache. More precisely, we can be sure about one invalid line in this set, which we assumed will be filled with preference once b is accessed. Therefore, when reaching c, we will find it still cached and account the access as a hit. More important, as long as we only consider private caches and exclude the lock itself, the cache access pattern follows exactly that of the application and there is no interference from concurrently executing applications.

So far we have only discussed a single level of private caches. However, it is easy to see that our approach extends to multiple levels of private caches if shared data is always invalidated from all these caches prior to releasing the data protecting lock.

3.2 Shared Caches

Cache level	clean	dirty
L1 hit	7 cycles	–
L1 miss / L2 hit	17 cycles	~34 cycles
L2 miss / L3 hit	56 cycles	~224 cycles
L3 miss	215 cycles	~1720 cycles

TABLE 1: *2.66 GHz Intel Core i7-920 quad core hit and miss latency per cache level*

In the previous section, we have seen for mutually exclusive accessed objects how to avoid the second source of unpredictability (see Section 2.1). We now focus on the first source. However, before we do so, let us quickly recapitulate on the need of predictable shared caches.

Column *clean* of Table 1 shows the cache access latency of an Intel Nehalem processor for each cache level. The column *dirty* takes into account the possibility of having to write back dirty data. For this worst-case estimate, we assume

1. that all write buffers are full,
2. that the fetched cacheline is allocated at each level,
3. that all these allocations cause the write back of dirty data and
4. that all these write backs miss in all caches up to the given level.

That is, for an L2 miss / L3 hit, we have to allocate the fetched line from L3 to L2, which causes a write back hitting in L3, and from L2 to L1, which triggers a write back to L2. This write back in turn triggers a second write back from L2 hitting in L3 to make room for the data that has to be fetched from L3. This totals to four L3 accesses. Notice write buffers between cache levels typically prevent applications from experiencing these high latencies. However, unless we can predict the eviction pattern of a cache, we have to assume pessimistically that these buffers are full. What do these numbers tell us?

3.2.1 Exclusive Data in Shared Caches

	value	ratio
Time	14.5 min	–
no. of memory accesses	720 billion	1
L2 evictions	23.4 billion	3.2%
L3 hit	16.1 billion	2.3%
L3 miss	1.5 billion	0.2%

TABLE 2: *Linux 3.9.11 kernel compile on a 2.66 GHz Intel Core i7-920 quad core. Ratio denotes the fraction of all data accesses that cause these events*[3].

To see whether predictable shared caches are required for exclusive data let us assume that caches are not inclusive and that there are no snoop filters in place that keep track of data located in private caches. Notice, this does not preclude inclusive shared caches (e.g., L4) and the presence of snoop filters to find cachelines in caches that are shared by some of the cores (e.g., L3). We merely require that inclusiveness is not extended to private caches because we maintain their consistency by other means.

Under these assumptions, hits and misses remain predictable up to the lowest level private cache (e.g., L2) and we are only uncertain whether a write back to L3 will actually hit or miss. Hence, when calculating the worst case access times for an application, we must assume an L3 miss at the cost of 215 cycles on the above i7-920 plus an L3 write back at another 215 cycles. To see the severity of such write backs, we have measured the number of L2 evictions, the number of L3 hits and misses and the total number of memory accesses hitting in L1 and L2 for a compile of the 3.9.11 Linux kernel (which admittedly is not what is usually considered a real-time application with well understood memory access pattern). Table 2 shows the results. The overall execution time of the compile run was about 875 seconds. Of the 720 billion memory accesses of the Linux compile, 97.6% could be served from L1 and L2, which in our setting are predictable. If we now turn all L2 evictions and other L3 accesses into L3 misses, the overall execution time is increased by approx. 16 minutes, which is just a little more than double the average case execution time for such a memory intensive tasks.

Notice that if it would be possible control the write policy of individual levels, we could improve on these numbers by setting all shared cache levels to write through and non write allocating.

3.2.2 Shared Data in Shared Caches

The primary role of shared caches for shared data is to allow for a fast exchange of information between cores. To exchange data, the sending core writes data from its private caches into the shared cache level from which the receiving core can retrieve it. Without shared caches, low-cost cache-to-cache transfers are only possible by pulling data directly from the sending core's private caches, invalidating it in the process, or by pushing data into the receiving core's cache, both obviously re-establish the interference we seek to avoid.

To facilitate fast exchange of shared data and in particular quick release of held locks, which in our setting requires writing back dirty data into the shared cache, we have to ensure that accessed lines remain cached whenever the access pattern allows so. From our assumption that shared objects are only accessed under lock protection, we know that other cores won't snoop out these lines while the lock is held. Consequently, we must only ensure that these lines are not lost due to eviction. More precisely, our goal is to prevent eviction as a result of other cores allocating cachelines. In particular, we do not seek to prevent self eviction of a core's own cachelines to not limit the amount of memory accessible in critical sections.

One solution well known in real-time systems is cache coloring, either at the level of individual cachelines or more coarse grain at the unit of individual pages [6]. The traditional use of cache colors is to partition the memory such that a single application allocates object only in a limited set of colors which is disjoint from the color sets assigned to other applications. Therefore, no matter how much cachelines the application accesses, it cannot evict the other application's cachelines. Our solution however, has to follow a fundamentally different coloring scheme. First, for exclusive data, all applications may share the same color because we have seen that exclusive data needs not to be protected from eviction in the shared caches. It is expected to reside in the private cache levels anyway. We assign one dedicated set of colors to hold all applications' exclusive data. Shared data has to remain in the cache if possible. Therefore, if a locked shared object consists of m cachelines of a particular color, we have to assign this color to the applications that share this object and make sure that if m is smaller than the number of ways n in the cache, then the same color is assigned to objects with at most $n - m$ cachelines in this color.

[3]Please note that L2 evictions and L3 hits and misses also consider instruction accesses whereas the number of memory accesses does not. We deliberately ignored instruction accesses because recent Intel cores fetch instructions in chunks of 16 bytes and interpret all instructions in this chunk at once.

In a sense colors become a resource, which when object locations can change (e.g., like in garbage collected languages) can be scheduled by allocating hot objects into reserved colors and cold objects into the colors used for exclusive data where no further guarantees are given.

4 Advanced Synchronization

Although an elaborate discussion of advanced synchronization protocols has to remain future work, we would like to sketch possible extensions of our work to other synchronization schemes.

4.1 Reader-Writer Locks

Our approach trivially extends to reader-writer locks. As long as the object is read only, multiple readers may simultaneously hold a copy in their private caches. Prior to releasing their lock, readers invalidate the shared object's cachelines. Therefore, when a writer obtains the lock, it will find no further valid copies and can modify the object at hand.

Notice, a general optimization for read-most locks or for locks that are frequently held by the same thread without other threads holding it in the mean time, is to invalidate cachelines lazily. That is when releasing the lock, cachelines are eagerly written-back to the shared cache levels but copies are kept in the core's private caches. Therefore, if the thread regains exclusive or read-shared access without intermittent modifications, it can simply continue. On the other hand, if it detects that the object is modified in the mean time, it has to first invalidate at least those cachelines of its private copy that have been modified to gain access to the most recent version in the shared cache. The eager write back ensures that none of these cachelines are written back, which could possibly cause more recent data in the shared caches to be overwritten.

4.2 RCU

Transactional schemes such as RCU and transactional memory are based on the assumption that objects can be kept invisible until they are committed. With RCU, this visibility making happens through an atomic update swinging an inbound pointer to the readily prepared object. At this time the new copy becomes visible and pending references to old copies are "garbage collected" by awaiting the next grace period. As the swing by marks the point where a copy becomes visible, this is the time by which the cachelines must be written back to the shared caches. While preparing the copy, cachelines can remain exclusive to the preparing core.

4.3 Transactional Memory

In addition to RCU, transactional memory facilitates intermittent modification detection and inplace modification. For as long as we can guarantee that no cacheline of the transaction is evicted from the private caches, which is possible as long as private caches remain predictable, it is possible to perform arbitrary speculative modifications. To commit these modifications however, we have to make sure to atomically commit this data if no modification has happened in the mean time. With snoop-based caches, this could be done by atomically discarding the indicators which mark the transaction state as speculative. This way the cache coherence protocol will automatically return the committed state. However, this again reintroduces the private cache interference we have avoided so far. An alternative is to version transactional objects and to lock this version number for the atomic commit and increase. That is, when starting a transaction, the version number is read (without acquiring a lock). Strictly after this increase, the object can be read and modification may start. Upon commit, the version number is locked, checked against the previously read version number and if both match the core has both read a consistent object and no non-speculative modifications have happened in the mean time. It is safe to commit the state by writing back all modifications while holding the lock. Finally, the version number is updated to a new value, which invalidates all intermediate transactions.

5 Evaluation

To obtain a first indication on the performance of our approach, we have implemented a simple lock-protected object and varied its size from one to eight cachelines. All lines have been allocated to the same color, varying colors over several runs of the experiments to exclude cross talk. Measurements were performed on a 2.66 GHz Intel i7-920 and on an ARM Cortex-A9 based dual-core system embedded in a Tegra2 evaluation platform.

For the benchmark, we allocated a spinlock in cache coherent memory. After acquiring the lock, each cacheline of the object is accessed exactly once before the object is evicted from the private caches

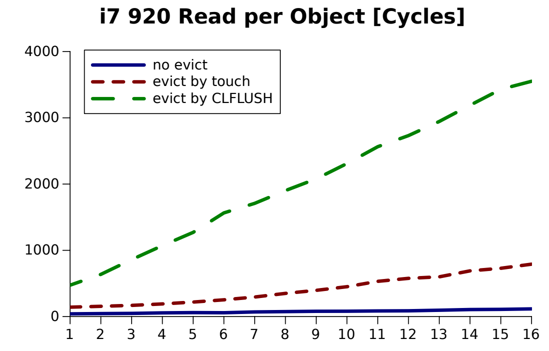

FIGURE 4: *Read costs for entire object.*

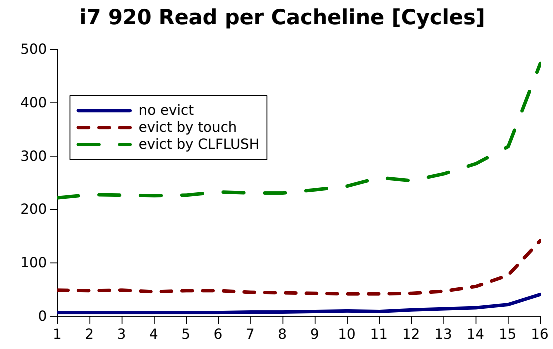

FIGURE 5: *Read costs average per cacheline.*

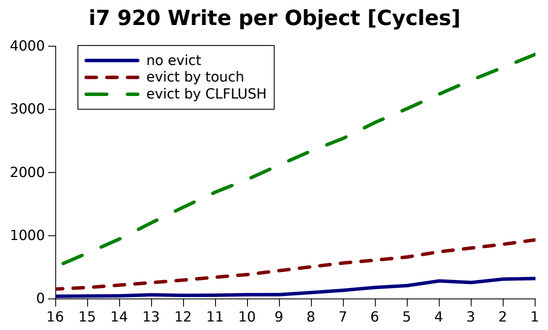

FIGURE 6: *Write costs for entire object.*

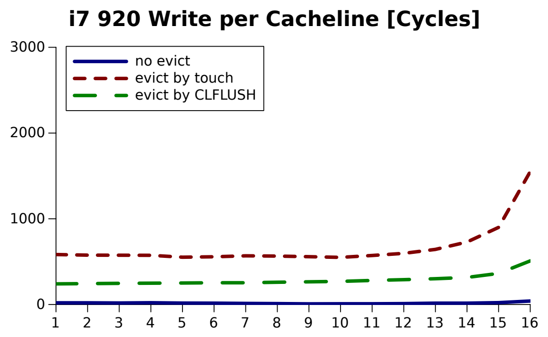

FIGURE 7: *Write costs average per cacheline.*

and the lock is released. Shown is the average execution time of this sequence as an indicator for the performance penalty due to selective invalidation and the object access time without any invalidation (i.e., leaving coherency up to the cache coherency protocol). The benchmark assumes repeated acquisition of the lock by one single core without interference from other cores.

5.1 x86

For the Intel i7, we have evaluated two eviction methods. The first exploits the fact that cache eviction follows a pseudo least recently used replacement policy by accessing first a number n of exclusive cachelines, which corresponds to the number of ways m in the lowest level private cache minus the size of the object in cachelines. Since our approach makes private caches predictable, a compiler can easily generate an access sequence for the last n exclusive cachelines cached in the relative order in which they have been accessed. Prior to loading these n lines, $m - n$ other cachelines of the same color were loaded. In combination, these m cachelines evict the shared object and ensure that the n exclusive lines are the youngest. Notice, on Intel processors, L2 typically acts as a pure victim cache. That is, upon a read miss in L1 and L2, L2 is bypassed and the cacheline is only allocated to L1. When later a line is evicted from L1 it drops to L2 from where it can be fetched when the application accesses it in the near future. The eviction sequence for recent Intel processors therefore has to include k additional accesses to exclusive memory, where k is the number of ways in L1. Again, because L1 is predictable, it is possible for a compiler to generate the required sequence while maintaining the relative age of exclusive lines.

In addition to this software only approach, we have also issued a cacheline flush instruction for each line of the shared object plus a memory fence to ensure that all write backs complete prior to releasing the lock. Unfortunately, x86 does not support an instruction that just invalidates cachelines from the private cache levels. Even worse, the `CLFLUSH` instruction broadcasts the invalidate to all cores in the coherence domain, evicting the cacheline also from the other cores' private caches.

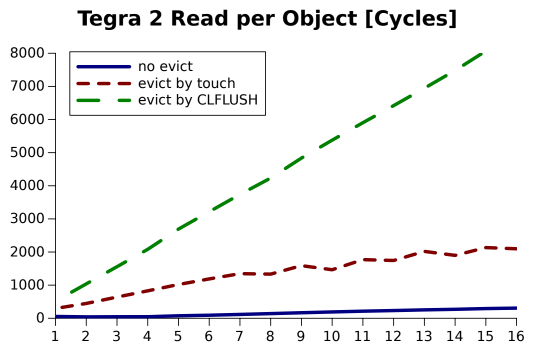

FIGURE 8: *Read costs for entire object.*

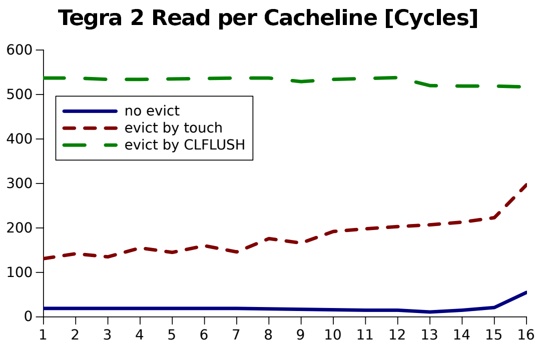

FIGURE 9: *Read costs average per cacheline.*

5.2 ARM

A hardware cache flush on ARM is not as far reaching as on x86. However, it is clearly not optimized for performance. Depending on the concrete ARM architecture, it may or may not be possible to evict caches in software. For example, ARM 11 MPCore processors implement FIFO replacement in their caches whereas ARM Cortex A9 adopts a pseudo random strategy.

5.3 Results

Figures 4–7 show the execution times of one critical section for increasing object sizes measured on an Intel i7-920-based system at 2.66 GHz. Figures 8–11 present the same measurements for a Tegra 2 ARM Cortex A9 system. The top two figures (Figure 4 and 6 for the Intel box and Figure 8 and 10 for ARM) show the overall execution time for the critical section. The bottom two figures show an average of these costs per cacheline. All measurements have been executed as Linux applications respectively in kernel mode for Tegra 2 to avoid the overhead of virtualizing the cache flush instruction. To better see the difference between manual eviction by touching all cachelines and no eviction, Figure 12 shows a more detailed view of the i920 read/write object times, excluding the CLFLUSH plot.

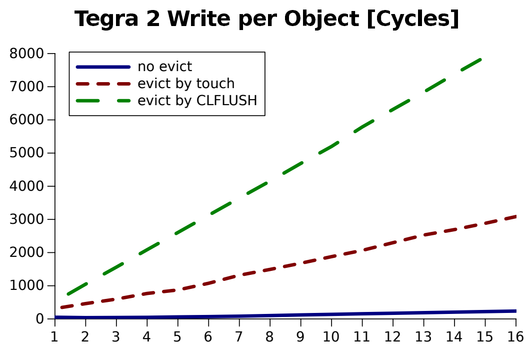

FIGURE 10: *Write costs for entire object.*

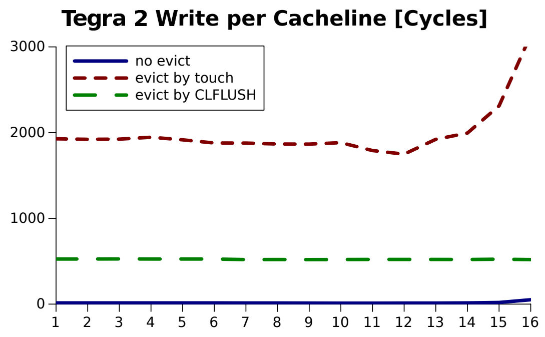

FIGURE 11: *Write costs average per cacheline.*

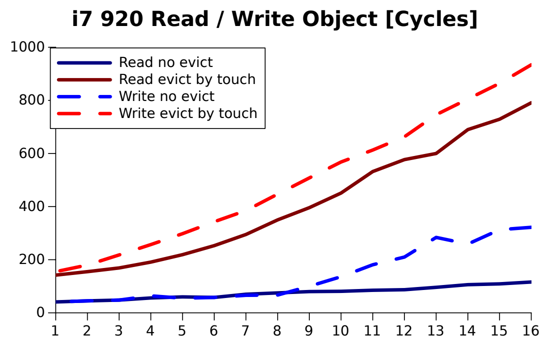

FIGURE 12: *Read/Write costs for entire object excluding CLFLUSH.*

As expected selective invalidation comes at an additional cost that increases with the size of the object and hence with the number of cachelines that have to be evicted from the private caches. However, we also see for the i7-920 that up to a size of 12 cachelines, the costs for invalidating larger objects amortize themselves when broken down per invalidated cacheline. With the exception of Figure 7, we also see that the costs for invalidating cachelines using the hardware instructions `CLFLUSH` or its ARM counterpart are much more expensive than a manual eviction by loading cachelines that collide with the cached data. Compared to the unpredictable case, we see a factor 10 performance loss for the critical section. The exclusive data access times are not affected by these measures.

Although much room for improvement remains, in particular in the form of better tailored cache flush instructions, we are confident that non-inclusive, non-coherent private caches are a first step towards more predictable many core systems.

6 Related Work

The questions of predictable caching has concerned many researchers before us. We already mentioned the work by Ferdinand and Wilhelm [1] and more generally the work of Prof. Reinhard Wilhelm and his group. Their focus is on the analytical side seeking to cope with whatever caches are around. Our approach comes with a suggestion of a new cache architecture where private caches are excluded from the coherence domain for the sake of predictability while maintaining shared caches for non-real-time applications and fast data exchange between cores. Wolfe [8] and later Liedtke et al. [6] at the operating-system level, introduce cache coloring as a means to partition to prevent one application from evicting the cachelines of the others. While certainly predictable, cache coloring falls short of increasing the predictability of shared data unless like in our work, precautions are in place to reduce the number of concurrent accesses.

Among others, Bastoni et al. [9] quantify the impact of cache related preemption delays, which Kim et al. [10] exploit to improve schedulability of multicore real-time systems. Rather than partitioning the cache for each application, they introduce per core partitions and consider worst-case preemption overheads when scheduling applications to these cores. The same analysis applies to our approach when assigning more than one application thread to a single core.

Our work extends prior work on on-demand coherent caches (ODC^2) by Uhrig et al. [11]. Like our approach, they propose flushing shared data prior to leaving the protected lock. However unlike our approach, they introduce specific hardware to perform this flush. Also, they are currently limited to L1 caches whereas our software-only approach extends to multiple cache levels as long as they are not inclusive or otherwise equipped with snoop filter logic. Moreover, because shared data is kept coherent, ODC^2 requires additional hardware support to extended to reader-writer or transactional synchronization schemes. The latter relies on shared data being speculatively written into the cache.

7 Conclusions

Introducing a software-only selective invalidation scheme for private caches, we have shown how to preserve predictability of exclusive and shared data accesses while maintaining fast core-to-core data exchange and coherent data sharing if it is possible to decouple private caches from the shared caches' coherence protocol. We have shown how a cache analysis by Ferdinand and Wilhelm can be extended to predict whether such accesses hit or miss in the cache and hence what their worst case latencies are.

Although the costs for selective invalidation are significant on today's desktop and embedded multicore systems (we measured a factor 10 performance penalty for short critical sections over small to medium sized shared objects), there remains a large potential for further performance optimization without sacrificing predictability of the multi core or fast core-to-core communication via shared caches.

References

[1] *C. Ferdinand and R. Wilhelm* "Fast and Efficient Cache Behavior Prediction" Real-Time Systems Journal, 131-181, vol. 17(2/3), 1999

[2] *T. Sondag and H. Rajan* "A Theory of Reads and Writes for Multi-level Caches" Technical Report #09-20b, Iowa State University, Apr. 2010

[3] *H. Cheong and A. Veidenbaum* "A Cache Coherence Scheme with Fast Selective Invalidation" 15th International Symposium on Computer Architecture, 1988, pp. 422–431

[4] *A. Pyka, M. Rohde and S. Uhrig* "A Real-Time Capable First-level Cache for Multi-cores"

Workshop on High-performance and Real-time Embedded Systems (HiRES 2013), January 23, 2013

[5] *C. Cullmann, C Ferdinand, G. Gebhard, D. Grund, C. Maiza (Burguire), J. Reineke, B. Triquet and R. Wilhelm* "Predictability Considerations in the Design of Multi-Core Embedded Systems" Journal Ingénieurs de l'Automobile, Sept. 2013, vol. 807

[6] *J. Liedtke, H. Härtig and M. Hohmuth*, "OS-Controlled Cache Predictability for Real-Time Systems" 3rd IEEE Realtime Technology and Applications Symposium (RTAS), June 1997, pg 213–223

[7] *N. Asmussen, H. Härtig and M. Völp*, "A Microkernel-based System for Minimalist Cores" 24th Symposium on Operating System Principles — Poster Session, Nov. 2013, Farmington, PA, USA

[8] *A. Wolfe*, "Software-based cache partitioning for real-time applications" 3rd International Workshop on Responsive Computer Systems, Sept. 1993

[9] *A. Bastoni, B. Brandenburg and J. H. Anderson*, "Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability" 10th Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT), 2010

[10] *H. Kim, A. Kandhalu and R. Rajkumar*, "A Coordinated Approach for Practical OS-Level Cache Management in Multi-Core Real-Time Systems" In Euromicro Conference on Real-Time Systems (ECRTS), 2013

[11] A. Pyka, M. Rohde, S. Uhrig and J. Fernandes "On Demand Coherent Cache for Parallelised Hard Real Time Applications" In Euromicro Conference on Real-Time Systems (ECRTS) — Work in progress, 2013