Selfish-LRU: Preemption-Aware Caching for Predictability and Performance

Jan Reineke*, Sebastian Altmeyer†, Daniel Grund‡, Sebastian Hahn*, and Claire Maiza§
*Saarland University, Saarbrücken, Germany
reineke@cs.uni-saarland.de, sebastian.hahn@cs.uni-saarland.de
†University of Amsterdam, Netherlands
altmeyer@uva.nl
‡Thales Germany, Business Unit Transportation Systems, Germany
daniel.grund@thalesgroup.com
§INP Grenoble, Verimag, Grenoble, France
claire.maiza@imag.fr

***Abstract*—We introduce Selfish-LRU, a variant of the LRU (least recently used) cache replacement policy that improves performance *and* predictability in preemptive scheduling scenarios.**

In multitasking systems with conventional caches, a single memory access by a preempting task can trigger a chain reaction leading to a large number of additional cache misses in the preempted task. Selfish-LRU prevents such chain reactions by first evicting cache blocks that do not belong to the currently active task. Simulations confirm that Selfish-LRU reduces the CRPD (cache-related preemption delay) as well as the overall number of cache misses. At the same time, it simplifies CRPD analysis and results in smaller CRPD bounds.

I. Introduction

Caches are an important part of the memory hierarchy of current processors. By providing low-latency access to a small part of the contents of main memory, they bridge the gap between low-latency cores and high-latency main memories. Which part of main memory to store in the cache is decided at runtime by the replacement policy. Even though caches complicate WCET (worst-case execution time) analysis, their use is compulsory in many real-time systems as they reduce execution times even in the worst case.

Preemptive scheduling is a mechanism to execute multiple tasks on a single-threaded processor, without requiring cooperation of the scheduled tasks. It is well-known in real-time scheduling that many task sets are only schedulable under a preemptive scheduling regime. This is the case in particular, when task sets include both high-frequency tasks with short deadlines and long-running tasks.

Thus, often, both caches and preemption are required to meet all deadlines of a set of real-time tasks. However, in contrast to simplifying assumptions in many scheduling analyses, preemption does not come for free. It takes time to perform a context switch from one task to another, and more subtly, the execution time of a task may increase due to preemptions. The additional execution time of a task that can be attributed to preemption is known as the *preemption delay*. The main contributor to the preemption delay are additional cache misses in the preempted task, due to memory accesses in the preempting tasks. The cost incurred by these additional cache misses is referred to as the CRPD (cache-related preemption delay).

There are two approaches to account for preemption delays: 1.) By conservatively taking into account the effect of all possible preemptions in the WCET bound [1]; or 2.) by creating a richer interface between WCET analysis and schedulability analysis. The first approach yields extremely pessimistic WCET bounds. To reduce pessimism, following the second approach, multiple new schedulability analyses [2] have been proposed recently. In addition to a bound on the WCET of a non-preempted execution of a task, such schedulability analyses require characterizations of the cache behavior of the tasks that are sufficient to bound the CRPD.

Following Liu et al. [3], one can distinguish two types of *context-switch misses*, i.e., cache misses that are a consequence of preemptions:

1) A cache miss in the preempted task is called a *replaced context-switch miss* if the requested memory block was replaced *during* the preemption.
2) Context-switch misses may replace additional memory blocks of the preempted task. Cache misses caused by accesses to such memory blocks are called *reordered context-switch misses*.

Consider a loop that references the memory blocks a, b, c, and d repeatedly. Then an LRU cache of size four will cycle through the following four cache states:

$$[d,c,b,a] \xrightarrow{a} [a,d,c,b] \xrightarrow{b} [b,a,d,c] \xrightarrow{c} [c,b,a,d] \xrightarrow{d} [d,c,b,a]$$

Except for the first loop iteration there will be no cache misses. Now, assume that the execution of the loop is preempted and the preempting task performs a single memory access to block $\boldsymbol{e}$ evicting block a and resulting in cache state $[\boldsymbol{e},d,c,b]$. In the first loop iteration after the preemption four context-switch misses occur:

$$[\boldsymbol{e},d,c,b] \xrightarrow{a^*} [a,\boldsymbol{e},d,c] \xrightarrow{b^\dagger} [b,a,\boldsymbol{e},d] \xrightarrow{c^\dagger} [c,b,a,\boldsymbol{e}] \xrightarrow{d^\dagger} [d,c,b,a]$$

The miss to a is a replaced miss, tagged with $*$. This triggers

the subsequent reordered misses to b, c, and d, tagged with †, which were *not* replaced during the preemption.

Such chain reactions causing reordered context-switch misses have been overlooked in previous CRPD analyses and in empirical evaluations of the context-switch cost, as observed in [4] and [3]. However, based on simulations, Liu et al. [3] report that reordered misses account for 10% to 28% of all context-switch misses.

We introduce a new replacement policy for multitasking systems, Selfish-LRU, a variant of LRU (least-recently used). In contrast to regular LRU, Selfish-LRU distinguishes cached memory blocks not only by the recency of their last access, but also by the task they belong to. Upon a miss, memory blocks belonging to the active task are prioritized over memory blocks belonging to inactive tasks.

Reconsider the example from above. In Selfish-LRU, after the preemption, the replaced miss to a occurs as in LRU. However, a evicts block $\boldsymbol{e}$ because it belongs to another task:

$$[\boldsymbol{e}, d, c, b] \xrightarrow{a^*} [a, d, c, b] \xrightarrow{b} [b, a, d, c] \xrightarrow{c} [c, b, a, d] \xrightarrow{d} [d, c, b, a]$$

The advantage of Selfish-LRU over LRU is that it completely eliminates reordered context-switch misses. Thereby, it reduces the context-switch cost *and* it simplifies static CRPD analysis.

In this paper, we make the following contributions:

1.) We introduce Selfish-LRU, a novel replacement policy targeted at preemptive systems that improves upon LRU, the most predictable replacement policy known to date, in terms of both *performance* and *predictability*.

2.) We show how state-of-the-art static CRPD analyses for LRU based on useful cache blocks (UCBs), evicting cache blocks (ECBs), and the notion of Resilience can be adapted to Selfish-LRU. In particular, the results of ECB analyses can simply be reinterpreted for Selfish-LRU and Resilience analysis is *simplified*. The resulting analyses provably yield *smaller* CRPD bounds.

3.) We empirically evaluate Selfish-LRU by simulation and static analysis on SCADE models and a set of tasks from the Mälardalen benchmark suite. The number of cache misses observed during simulation and the CRPD bounds are *reduced* by up to 39% and 63%, respectively.

II. Selfish-LRU: Useful Properties and Formal Specification

In this section, we provide the intuition behind Selfish-LRU followed by a formal specification and a discussion of its useful properties. This concerns only the logical behavior, i.e., when and which memory blocks are replaced, not how this is realized in hardware, which will be discussed in the following section.

A. Intuition Behind Selfish-LRU

Caches exploit spatial and temporal locality. The intuition behind LRU is that, due to temporal locality, a memory block that has been used more recently than another block is also more likely to be used again soon. In most scenarios, this intuition is correct. However, *following* a context-switch, it can be wrong. Then, intuitively, memory blocks of the active task are more likely to be reused than blocks belonging to other tasks, even though those might have been used more recently. Selfish-LRU follows this intuition and prioritizes blocks of the active task; i.e., Selfish-LRU preferably replaces blocks belonging to inactive tasks upon a cache miss.

B. Formalization of States and Updates

In a set-associative cache, individual cache sets can be seen as fully-associative caches, and different cache sets are logically independent. Thus, we limit our description to that of fully-associative caches.

A fully-associative cache consists of k cache lines, where k is the associativity of the cache. In a regular cache, a cache line consists of the tag of the cached memory block, to identify which block's data is cached, and the cached data itself. Here, we are only concerned with which block $b \in B$ is cached, and not with the data stored in the cache line. In Selfish-LRU, in addition to the tag, we also need to keep track of the ID of the task $t \in T$ that a cache line belongs to. We define the set of logical Selfish-LRU states as

$$Q := ((B \times T) \cup \{\bot\})^k \tag{1}$$

A cache line is either invalid, denoted by $\bot$, or it contains a memory block $b \in B$ belonging to a task $t \in T$. Cache lines are ordered by the recency of their last access, from most- to least-recently used.

An access a consists of a memory block $b \in B$ and the ID of the active task $t \in T$. In the following, let $a = (b, t)$. Similarly, for each cache line l_i, $l_i = (b_i, t_i)$ or $l_i = \bot$. Then, a Selfish-LRU state is updated upon an access a as follows:

$$up([l_1, \ldots, l_k], a) := \begin{cases} [a, l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_k] & \textit{if } b_i = b \quad \textit{'hit'} \\ [a, l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_k] & \textit{elsif } l_i = \bot \quad \textit{'miss, invalid'} \\ [a, l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_k] & \textit{elsif } t_i \neq t \wedge \forall j > i \colon t_j = t \\ & \quad \textit{'miss, other'} \\ [a, l_1, \ldots, l_{k-1}] & \textit{elsif } \forall i \colon t_i = t \quad \textit{'miss, own'} \end{cases}$$

The update distinguishes four cases, described below:

1) 'Cache hit': The accessed block b is contained in the cache. In this case two things happen: The block becomes the most-recently-used (MRU). The 'ownership' of the block changes if the block previously belonged to a task other than the active one.

2,3,4) 'Cache miss': In any case, the newly inserted element assumes the MRU position. There are three different kinds of cache misses:
   2) The cache contains invalid lines: These are filled first.
   3) The cache contains blocks that do not belong to the active task: Selfish-LRU replaces the least-recently-used of these blocks.
   4) The cache consists of memory blocks of the active task only: Then, the least-recently-used block is replaced.

C. Useful Properties of Selfish-LRU

After providing a formal specification, let us state two useful properties of Selfish-LRU that can be exploited in WCET and CRPD analyses.

Property 1 (Non-preempted Execution): If a task is not preempted, it experiences the same cache behavior in Selfish-LRU as it does in LRU.

This means that known precise and efficient cache analyses [5] for LRU can be applied to Selfish-LRU during WCET analysis. LRU is generally considered to be the most predictable replacement policy in the non-preemptive scenario [6].

While Selfish-LRU cannot prevent *replaced context-switch misses*, it does eliminate *reordering context-switch misses*:

Property 2 (No Reordering Misses): Selfish-LRU does not exhibit *reordering context-switch misses*.

To see why this is the case, notice that Selfish-LRU will only replace a memory block of the active task if *no* blocks of other tasks remain in the cache. Thus, whenever a block of the active task is replaced, this would also have happened if the active task had run in isolation. As we will see in Section IV, due to Property 2, Selfish-LRU simplifies two state-of-the-art CRPD analysis approaches.

Note that any replacement policy that does not distinguish blocks of different tasks, e.g., PLRU or FIFO, exhibits reordering context-switch misses.

A further observation is that Selfish-LRU satisfies the *inclusion property* [7]. While, this is not relevant for WCET or CRPD analysis, it implies that the performance of Selfish-LRU can be efficiently determined for a range of associativities in a single simulation run:

Property 3 (Inclusion Property): Selfish-LRU is a *stack algorithm* [7]. Thus, it satisfies the *inclusion property*.

See the appendix for a proof sketch.

III. Implementation of Selfish-LRU

In this section we describe how Selfish-LRU can be implemented. We do so by going through Selfish-LRU requirements that go beyond those of a conventional system featuring regular LRU replacement and describe possible adaptations to meet those requirements.

A. Assignment and Maintenance of Task IDs

One requirement of Selfish-LRU is that each task needs to be assigned a unique ID, TID for short. This can be accomplished by adapting the (operating) system primitives offered for task creation. In addition to handling the usual parameters, e.g., code entry point, period, deadline, and priority, the primitive assigns consecutive TIDs to each newly created task.

The TID of the active task needs to be available to the cache. This can be accomplished by introducing an additional hardware register, the TID register. Similarly to other registers, the content of the TID register needs to be saved upon a context switch so that it can be restored when switching back into that context.

Special attention is required concerning the operating system itself as it needs to run with its own unique TID. Otherwise, the instructions and data referenced by the operating system would be cached using the TID of the previously active task. Hence, the hardware mechanism that saves the instruction pointer (IP) upon kernel mode entry (or upon interrupts) needs to be extended to additionally save the TID register. In case the 'operating system' is rather small, e.g., when it only comprises an interrupt handler and a scheduler, it might be sensible to simply disable caching during that time. In such a case, saving and restoring the TID register can also be implemented in software.

B. Cache Implementation

In the following, we discuss the implementation of Selfish-LRU in a physically-tagged, physically-indexed cache, which is common in embedded systems.

Logically, a cache consists of n cache sets, each of which consists of k cache lines, each of which comprises the actual data, some status bits, and a tag to identify the block stored in that cache line. For Selfish-LRU, each cache line is additionally assigned a TID that indicates the owner of the data, i.e., the task that last referenced that data.

As opposed to the logical structure of a cache ('array of structs'), the physical implementation ('struct of arrays') has to serve efficiency goals: To enable efficient tag lookups and comparisons, cache tags are stored separately in tag RAMs, one tag RAM per cache way. This way, all tags of a cache set can be accessed simultaneously, which is required to decide quickly whether a memory access constitutes a hit or a miss, and to locate the cached data upon a hit. We propose to store the TID of each cache line together with its tag in the tag RAMs.

Upon each cache access, the active task needs to take the ownership of the referenced cache line, i.e., the TID of the cache line needs to be set to the TID stored in the TID register. In case of a cache miss, there is no overhead in terms of additional circuit latency as the new tag needs to be written anyway. In case of a cache hit, compared to LRU, Selfish-LRU requires an additional write operation to the tag RAM. However, this write can happen in parallel to the state update of the LRU policy, which needs to be performed on each access anyway. The data to be written, i.e., tag and TID, can be assembled in parallel to the cache lookup.

The overhead of the additional TID storage can be estimated roughly as follows. The tag size of a cache with 32 bytes line size and 128 cache sets is 20 bits – assuming a 32 bit address space. Neglecting status bits, a single cache line therefore requires $32 \cdot 8 + 20 = 276$ bits. The overhead for task sets comprising less than 256 tasks, i.e., 8 bit per TID, thus is less than $8/276 \approx 2.9\%$. Even if that amount could be completely utilized for the net capacity, the benefit would likely be marginal.

What remains to be discussed is the implementation of Selfish-LRU's replacement decisions. Sudarshan et al. [8] present different implementations of LRU. While LRU needs to simply determine the least-recently-used cache line, Selfish-LRU needs to additionally take the task IDs into account.

Consider an LRU implementation that maintains *ages*: Each cache line is assigned an age between 0 and $k-1$, which is

updated on each access. Here k denotes the associativity of the cache. The age of a cache line l is the number of distinct cache lines of the same cache set that have been accessed after the last access to l. So, the age of the most-recently-used line is 0 and the age of the least-recently-used line is $k-1$. Upon a cache miss, the oldest line is chosen as the victim. Finding the oldest line can be implemented efficiently by a circuit for maximum value determination [9]. For Selfish-LRU, those ages can be augmented by a most-significant bit – before the oldest line is determined. That bit is set to 0 if the TID of the active task coincides with the TID of the cache line. Otherwise that bit is set to 1. This way, the 'oldest' cache line belongs to other tasks – or, in case there are only cache lines of the active task, the 'oldest' line is indeed the oldest line. Exactly as demanded by the Selfish-LRU specification.

Last but not least, let us point out similarities to the homonym problem that occurs in virtual memory systems: A virtual address is a homonym if it refers to two different physical addresses in different virtual address spaces. This can cause inconsistencies of cached data in case of virtually-addressed caches. One way to avoid this problem is to extend cache tags by address space IDs, which are almost identical to task IDs. This confirms that augmenting cache tags is viable.

IV. CRPD Analysis for Selfish-LRU

First note that static LRU cache analysis as part of WCET analysis does not have to be adapted to Selfish-LRU: as stated in Property 1, in non-preempted executions, Selfish-LRU behaves precisely like LRU.

CRPD analyses, on the other hand, either can be simplified, or can be used to deliver improved bounds on the preemption delay, as discussed below.

There are two main approaches to statically bound the CRPD:

1.) By analyzing the preempted task [10], [11], [12], [13], [14]: Additional misses can only occur for *useful cache blocks* (UCBs), i.e., blocks that may be cached and that may be reused later, resulting in cache hits. Static analyses have been proposed to safely approximate the set of UCBs.

2.) By analyzing the preempting task [15], [11], [12], [13]: The preempting task may only cause additional cache misses in those cache sets that it modifies. Thus, analyses to compute bounds on the number of *evicting cache blocks* (ECBs) have been developed. However, for set-associative caches, the approaches based on ECBs have so far been either imprecise [4] or unsound [12], as shown in [4].

Recently, Altmeyer et al. [16] introduced the notion of *resilience* of cached blocks. The resilience of a useful cache block is the amount of 'disturbance' by a preempting task that the block may endure before becoming useless to the preempted task. Resilience analysis computes lower bounds on the resilience of each useful cache block. These lower bounds can then be combined with upper bounds on the evicting cache blocks to determine a set of useful cache blocks that are guaranteed to *remain useful* after the preemption.

LRU is by far the best understood replacement policy in terms of CRPD analysis. In fact, Burguière et al. demonstrated that neither the number of ECBs nor the number of UCBs can be used to safely bound the CRPD [4]. Thus, the approaches described above apply to LRU only. Apart from the approach sketched in [4] no CRPD analyses for policies other than LRU, such as FIFO or PLRU, are known.

It is also important to mention that sound approaches to bounding the CRPD exist only for *timing compositional* architectures [17], [18], in which the cost of any additional cache miss can be bounded by a constant number of execution cycles. This issue is orthogonal to the choice of replacement policy and thus applies to LRU and Selfish-LRU alike.

A. Bounding the CRPD using UCBs

As for LRU, the number of useful cache blocks (UCBs) of the preempted task, as defined by Lee et al. [19], is a bound on the number of additional cache misses for Selfish-LRU. The same holds for definitely-cached useful cache blocks (DC-UCB) as defined in [14]. The following formula denotes an upper bound on the CRPD based on useful cache blocks, where BRT is the block reload time, n is the number of cache sets, and $\mathrm{UCB}(s)$ the set of UCBs in cache set s:

$$\mathrm{CRPD}^{\mathrm{SLRU}}_{\mathrm{UCB}} = \mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{UCB}} = \mathrm{BRT} \cdot \sum_{s=1}^{n} |\mathrm{UCB}(s)| \,. \tag{2}$$

Note that this bound (and all the following bounds) is (are) computed for each program point of the task. A CRPD bound for the entire task is then given by the maximum CRPD bound over all program points.

B. Bounding the CRPD using ECBs

In contrast to bounds based on UCBs, Selfish-LRU improves the CRPD bound that can be derived from the set of evicting cache blocks (ECBs) of the preempting task. Somewhat obviously, the number of ECBs bounds the number of *replaced context-switch misses*. In case of LRU, as discussed in the introduction, a single replaced miss may trigger $k-1$ reordered misses, where k is the associativity of the cache. For LRU, the following formula therefore captures the best bound on the CRPD based on ECBs:

$$\mathrm{CRPD}^{\mathrm{LRU}}_{\mathrm{ECB}} = \mathrm{BRT} \cdot \sum_{s=1}^{n} \begin{cases} 0 & \textit{if } |\mathrm{ECB}(s)| = 0 \\ k & \textit{otherwise} \end{cases} \tag{3}$$

For Selfish-LRU, as stated in Property 2, no *reordered context-switch misses* are possible. Thus, we get the following improved bound:

$$\mathrm{CRPD}^{\mathrm{SLRU}}_{\mathrm{ECB}} = \mathrm{BRT} \cdot \sum_{s=1}^{n} |\mathrm{ECB}(s)|. \tag{4}$$

As an immediate consequence, we also get an improved CRPD bound based on the ECBs of the preempting task and the UCBs of the preempted task:

$$\mathrm{CRPD}^{\mathrm{SLRU}}_{\mathrm{UCB\&ECB}} = \mathrm{BRT} \cdot \sum_{s=1}^{n} \min(|\mathrm{ECB}(s)|, |\mathrm{UCB}(s)|). \tag{5}$$

C. Bounding the CRPD using Resilience

The possibility of reordered context-switch misses complicates the computation of the resilience of useful cache blocks in case of LRU. See Altmeyer et al. [16] for a detailed description. As only replaced context-switch misses are possible in case of Selfish-LRU, resilience analysis is greatly simplified. The resilience of a useful cache block is determined by its age, i.e., its logical position in the cache. For example, the resilience of the most-recently-used block (age 0) is $k-1$, as the block will still be cached after accessing additional $k-1$ distinct memory blocks mapping to the same cache set. At the other extreme, the resilience of the least-recently-used block (age $k-1$) is 0. In general, the resilience of a block of age a is $k-a-1$. This holds, with the exception of shared memory blocks, i.e., blocks that are possibly accessed by other tasks. If such blocks are accessed by the preempting task, then they can be replaced immediately after resumption of the preempted task, as they now belong to a different task. The resilience of such blocks can be conservatively approximated by 0.

Must cache analysis [5], a static cache analysis based on abstract interpretation, computes upper bounds on the ages of memory blocks at all program points. It has been shown to be precise and to scale to large programs, and is in regular use in the commercial AIT WCET analyzer by ABSINT. Given an upper bound on the age a of memory block b one can compute a lower bounds $res(b)$ on the block's resilience. Shared data needs to either be specified by users or detected statically.

Together with information about the number of evicting cache blocks, $|\text{ECB}(s)|$, of the preempting task, resilience bounds allow to determine a subset of the useful cache blocks, $\text{UCB}(s)$, of the preempted task that must remain useful, and can thus not contribute to the CRPD [16]:

$$\text{CRPD}_{\text{RES}}^{\text{SLRU}} = \text{BRT} \cdot \sum_{s=1}^{n} \left| \overbrace{\text{UCB}(s)}^{\text{may be useful}} \setminus \overbrace{\{b \in B \mid res(b) \geq |\text{ECB}(s)|\}}^{\text{must remain useful}} \right|, \quad \overbrace{\phantom{\{b \in B \mid res(b) \geq |\text{ECB}(s)|\}}}^{\text{blocks that may have to be reloaded}}$$

Again, one can use DC-UCB(s) in place of UCB(s). Nested and multiple preemptions can be accounted for as described in Altmeyer et al. [16].

V. RELATED WORK

Lately, multiple efforts have been undertaken to develop microarchitectures that reconcile predictability with performance. Work along these lines includes classifications of existing microarchitectures in terms of their predictability [17], [20], studies of the predictability of caches [6], and proposals of new microarchitectural techniques, such as novel multithreaded architectures that eliminate interference between threads [21], [22], [23], [24], [25] and DRAM controllers that allow multiple tasks to share DRAM devices in a predictable and composable fashion [26], [27]. In the following, we review work specifically concerning the interplay between multitasking and the memory hierarchy.

Initially meant to reduce power consumption of an embedded device [28], today, *scratchpads* are advocated as means to increase a system's predictability. Similar to caches, scratchpads are small but fast memories located close to the processor. However, contrary to the dynamic behavior of caches, the decision which data to store in a scratchpad is taken statically. Hence, scratchpads are not transparent to a system designer (in contrast to caches) and result in non-uniform access latencies across the address space (like caches). The static nature of scratchpads hinders an efficient use in highly dynamic preemptive systems. Scratchpads have so far been used preferably in static non-preemptive systems. Only very recently, new implementations were proposed that allow efficient use of scratchpads in preemptive systems [29]. In the approach by Whitham and Audsley [29] called Carousel, similar to the handling of callee-save registers in function calls, the preempting task is responsible for restoring the state of the scratchpad it encountered at the point of preemption. Thus, the amount of data that needs to be saved and restored is proportional to the amount of scratchpad memory the preempting task intends to use. This corresponds to CRPD analyses based on ECBs, considering only the preempting task, which can be inefficient. The advantage of performing the save and restore operations in single DMA operations, as done in Carousel, is that transferring a large chunk of data at once can be more efficient than performing the same transfer in many small pieces as it would be done by a cache that is gradually refilled.

Cache partitioning [30], [31] and *cache locking* [32], [33] are techniques to reduce or even completely avoid cache-related preemption delay. In cache partitioning, each task is assigned a dedicated part of the cache to guarantee that a preempting task cannot evict cache blocks of other tasks. Partitioning can be implemented either in hardware, by means of a memory management unit, or in software, with the help of adapted compilers. The latter often results in substantial changes to the code and data layout so as to assert that each task only accesses its own partition [30]. In cache locking [33], cache lines are locked, so that a preempting task cannot evict their data. In *static* cache locking, data is loaded into the cache and locked for the entire task execution while *dynamic* cache locking only locks data during some predefined code regions. Both techniques trade *inter-task* for *intra-task* cache conflicts, i.e., they pay for the reduced cache-related preemption delay with a possible increase of the worst-case execution times.

Tan and Mooney [34] present a WCRT analysis for their previously proposed *prioritized cache*. Such a cache maintains a priority level for each cache way. A task T can only allocate a cache line in a cache way w if its own priority $prio(T)$ is greater than or equal to the priority of the cache way $prio(w)$. If it does so, the priority of the cache way rises to that of the allocating task; $prio(w) := prio(T)$. After a while, blocks of high-priority tasks therefore occupy the cache, even if they run only shortly and seldom. Low-priority tasks may as a consequence starve for cache space. With Selfish-LRU in contrast, each task can profit from the whole cache capacity.

TABLE I
BRIEF SUMMARY OF THE BENCHMARKS. M: MÄLARDALEN, S: EXAMPLES FROM THE SCADE SUITE, O: OWN SCADE MODELS. |ECB|: BOUND ON NUMBER OF EVICTING CACHE BLOCKS.

Source, Name	Size [byte]	|ECB|	Brief description
M, adpcm	25,275	>256	Adaptive pulse code modulation
M, compress	13,498	170	Data compression algorithm
M, edn	10,963	>256	Finite impulse response filter
M, statemate	52,513	>256	Statechart implementation of a car window lift control
S, cruisecontrol	46,275	107	Part of a cruise control system
S, flightcontrol	157,054	>256	Flight control system
S, pilot	58,948	94	Navigation system
S, stopwatch	32,066	150	Implementation of a stopwatch
O, lift	50,911	122	Elevator simulation
O, robodog	79,227	>256	Robotics system

VI. EXPERIMENTAL EVALUATION

In this section we present empirical evidence concerning the *runtime performance* and the *provable performance* of Selfish-LRU systems. Our evaluation target was to compare a Selfish-LRU system to an otherwise identical LRU system. In addition, we compare Selfish-LRU to a way-based partitioning approach.

A. Setup and Benchmarks

To determine the runtime performance, we implemented the LRU and Selfish-LRU replacement policies in the MPARM[1] simulation framework, specifically the ARMv7 processor model. On top of this simulator, we employed the RTEMS[2] operating system, which implements rate-monotonic preemptive scheduling. The simulation routine counts the number of cache misses on a per-task basis, so that cache accesses of the preempted task and the preempting task can be separated. We determine the number of context-switch misses by taking the difference between the overall number of cache misses of a task when preempted and the overall number of cache misses of the same task when executed without preemption.

To determine the provable performance, we implemented CRPD analyses based on UCBs, ECBs, and Resilience for both LRU and Selfish-LRU in the ARMv7 version of the commercial AIT WCET analyzer by ABSINT. This allows us to derive safe bounds on the context-switch misses based on the analysis of the preempting task, the preempted task and a combination of the two.

To evaluate way-based partitioning approach, we determine the runtime of each task for all possible partition sizes p, i.e., $p \in \{1, \ldots, \text{associativity} - 1\}$ when executed in isolation. The obtained values allow us to determine the optimal cache partitioning for a given set of tasks. Again, the simulation results are obtained using the MPARM simulator, whereas the analysis results are determined using the AIT WCET analyzer.

As benchmarks, we selected (a) four of the largest programs containing loops from the Mälardalen WCET benchmark suite[3]; (b) four SCADE models that come along with the SCADE distribution; and (c) two SCADE models developed as part of an embedded systems course. A brief description of the benchmarks is given in Table I.

Due to the relatively small size of the benchmarks used, we chose appropriate cache configurations with associativity $k \in \{4, 8\}$ and a number of cache sets $n \in \{32, 64, 128\}$. All caches feature a line size of 16 bytes and their total capacity C varies between 2 and 8 KiB.

B. LRU versus Selfish-LRU

In this section, we discuss the simulation and analysis results for caches with LRU and Selfish-LRU replacement policy.

a) Simulation Results: Figure 1 shows the effect of a single preemption by one of the smallest tasks, `pilot`. In each case, the number of additional misses under Selfish-LRU is lower than that of LRU. At twice the cache size, shown in Figure 2, most tasks fit into the cache together and the number of context-switch misses drops.

Of course, one cannot expect that Selfish-LRU improves all situations: When the preempting task has more ECBs, e.g., `edn`, the number of context-switch misses increases, see Figure 3. This increase is mainly due to more replaced misses; the percentage of reordering misses, which can be prevented by Selfish-LRU, is lower. When evicting the whole cache, e.g., with a preemption by `flightcontrol`, LRU and Selfish-LRU behave the same and are therefore not shown in a graph.

b) Analysis Results: As opposed to the simulation, where the context-switch misses for a limited set of different scenarios are measured, the analysis has to derive conservative upper bounds valid for *all* possible cases, i.e., *all* possible preemption points within the preempted task and also *all* possible execution traces of the preempting task. This, together with pessimism introduced by static analysis, explains the differences between the simulation and the analysis results. Consider Figure 6 for instance: The analysis determined an upper bound of 256 ECBs for `edn`, which are sufficient to evict the whole cache. Based on this information, it is impossible for any subsequent analysis to show an improvement of Selfish-LRU over LRU. Regarding Figure 3, either the simulations did not exhibit the worst-case behavior or the true number of ECBs is less than 256.

The task `pilot` has 94 ECBs. As Selfish-LRU prevents any reordering misses, 94 is thus an upper bound on the number of context-switch/replaced misses. This is confirmed by the analysis: see Figures 4 and 5. Often, a better bound than 94 can be obtained taking into account the resilience of a task's useful cache blocks, which is higher than in the case of LRU.

Overall, the analysis mostly confirms the results of the simulations. LRU and Selfish-LRU behave the same if *no* cache lines or if *all* cache lines are evicted during preemption. For preempting tasks of small or medium size, Selfish-LRU improves upon

[1] http://www-micrel.deis.unibo.it/sitonew/research/mparm.html
[2] http://www.rtems.com
[3] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

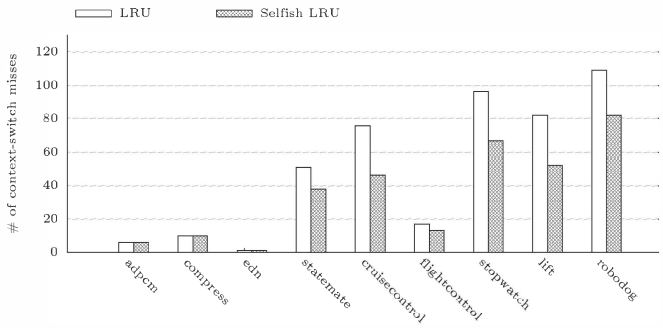

Fig. 1. Measured number of context-switch misses for each benchmark when preempted by `pilot`. $k = 4$, $n = 32$, and thus $C = 2$ KiB.

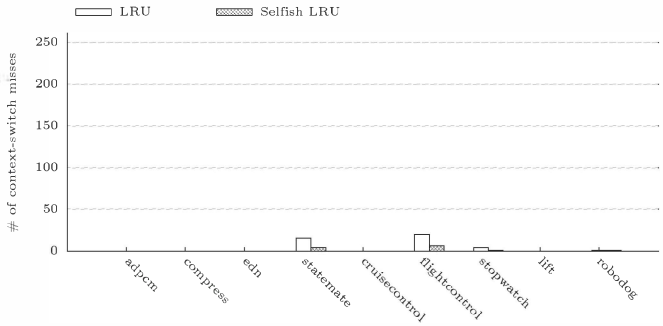

Fig. 2. Measured number of context-switch misses for each benchmark when preempted by `pilot`. $k = 8$, $n = 32$, and thus $C = 4$ KiB.

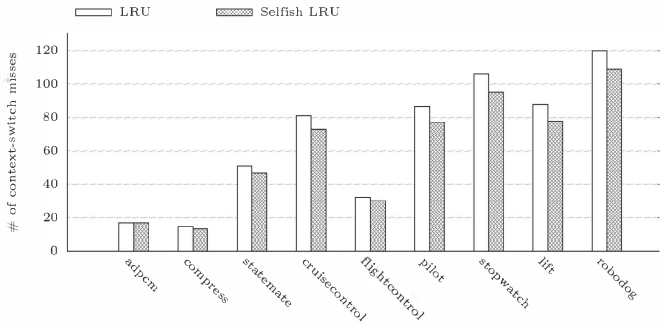

Fig. 3. Measured number of context-switch misses for each benchmark when preempted by `edn`. $k = 4$, $n = 32$, and thus $C = 2$ KiB.

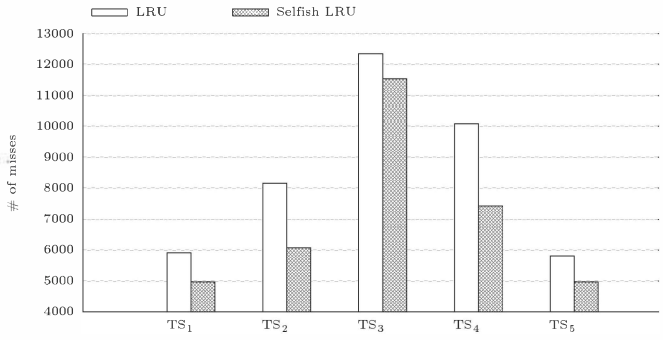

Fig. 7. Observed misses during the execution of different task sets. $k = 4$, $n = 128$, and thus $C = 8$ KiB.

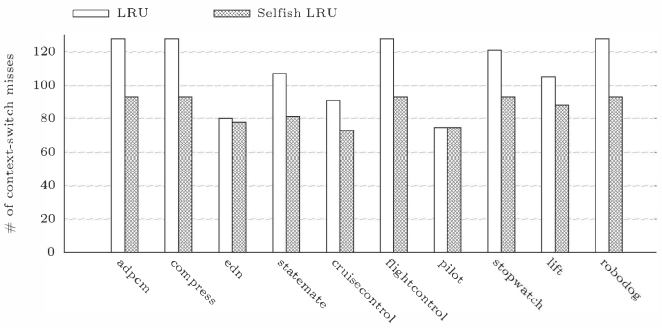

Fig. 4. Bounds on the number of context-switch misses when preempted by `pilot`. $k = 4$, $n = 32$, and thus $C = 2$ KiB determined by static analysis.

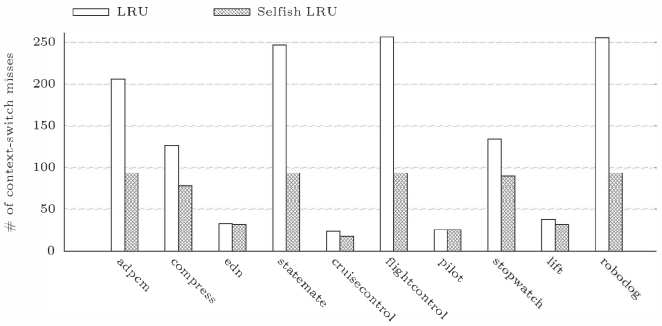

Fig. 5. Bounds on the number of context-switch misses when preempted by `pilot`. $k = 8$, $n = 32$, and thus $C = 4$ KiB determined by static analysis.

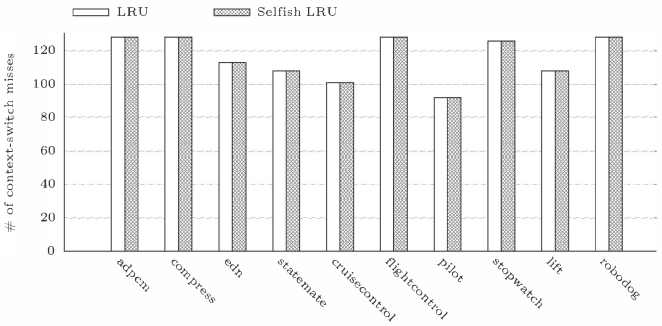

Fig. 6. Bounds on the number of context-switch misses when preempted by `edn`. $k = 4$, $n = 32$, and thus $C = 2$ KiB determined by static analysis.

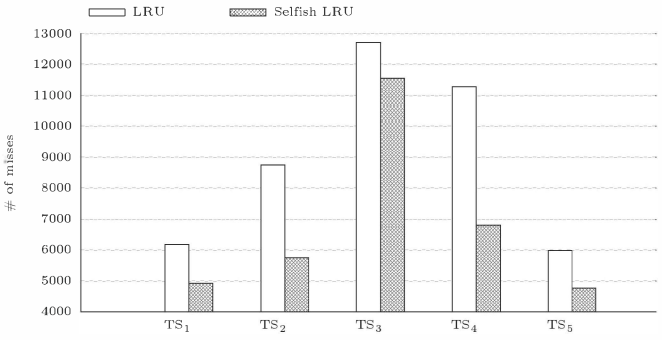

Fig. 8. Observed misses during the execution of different task sets. $k = 8$, $n = 64$, and thus $C = 8$ KiB.

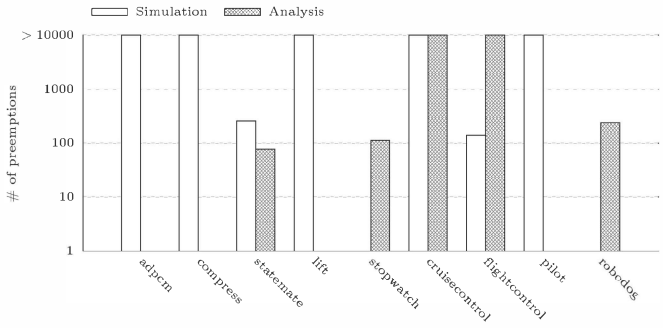

Fig. 9. Partitioning vs. Selfish-LRU. Simulation and analysis results for each benchmark when preempted by `edn`. $k = 8$, $n = 32$, and thus $C = 4$ KiB.

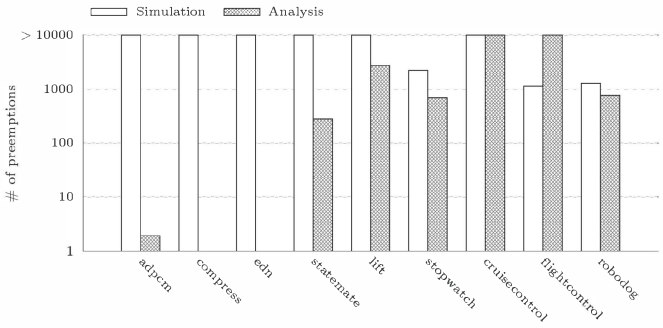

Fig. 10. Partitioning vs. Selfish-LRU. Simulation and analysis results for each benchmark when preempted by `pilot`. $k = 8$, $n = 32$, and thus $C = 4$ KiB.

LRU (see Figure 4), especially for larger caches (see Figure 5). In those scenarios, reordering misses constitute a large portion of all context-switch misses for LRU, which are eliminated in Selfish-LRU. Note that preemption by small tasks or interrupts with high priority is common to many real-time systems.

c) Complex Scenarios: Other, more complex scenarios are considered in Figure 7 and 8. They show the results for five task sets, each comprising 5 tasks taken from our set of benchmarks described in Table I. Each task set was subjected to the scheduling decisions of RTEMS, which include multiple preemptions per job as well as nested preemptions. The periods of the tasks vary between 15 and 150 time units and their phases, i.e., starting times relative to period, range from 0 to 50 time units. Compared to the single-preemption scenarios, these tasks sets are larger. Hence we benchmarked cache configurations with associativities 4 and 8, number of sets in $\{32, 64, 128\}$, and a line size of 16 bytes. The bars show the total number of misses incurred during five hyperperiods for LRU and Selfish-LRU, respectively.

For the smaller cache configurations, the difference between LRU and Selfish-LRU is only marginal: Due to multiple and nested preemptions, the cached memory blocks of a preempted task are often completely replaced by memory blocks of the preempting tasks. Therefore, Selfish-LRU cannot improve much over LRU in these scenarios.

However, for larger caches, the cached memory blocks are not replaced completely on (multiple and nested) preemptions. The overall performance improves significantly using the Selfish-LRU replacement policy, as depicted in Figure 7 and Figure 8. This confirms the utility of Selfish-LRU in preemptively scheduled systems.

C. Selfish-LRU versus Partitioning

As discussed in the related work section, cache partitioning eliminates inter-task interference at the expense of increased intra-task interference. In this section, we evaluate in which situations partitioning outperforms Selfish-LRU and vice versa. We consider *way-based* partitioning realized in hardware, as it is easy to analyze and does not require modifications to the software, which could influence the results in one direction or the other. As one increases the number of tasks sharing a partitioned cache, an individual task's partition becomes smaller and its intra-task interference increases. We bias the analysis in favor of partitioning by only considering simple scenarios consisting of two tasks, one low and one high priority task.

Let $time_A(p)$ and $time_B(p)$ denote the execution time of task A and B, respectively, given p cache ways of a k-way associative cache. Then $time^{Part}_{A,B}(n)$, as defined below, determines the time required to execute the lower priority task A once and the higher priority task B n times using the best possible partition p of the cache:

$$time^{Part}_{A,B}(n) := \min_{p \in \{1,\ldots,k-1\}} \{time_A(p) + n \cdot time_B(k-p)\}.$$

In the unpartitioned case with Selfish-LRU replacement, the time to execute the same scenario is determined as follows:

$$time^{CRPD}_{A,B}(n) := time_A(k) + n \cdot (time_B(\mathrm{A}) + CRPD_{A,B}(k)).$$

For increasing values of n the influence of the first summand in both $time^{Part}_{A,B}(n)$ and $time^{CRPD}_{A,B}(n)$ decreases and partitioning may become more and more beneficial.

For each pair of tasks A, B we determine the minimal n for which $time^{Part}_{A,B}(n) \leq time^{CRPD}_{A,B}(n)$. In Figures 9 and 10 we show this minimal n for execution times and preemption delays determined by simulation as well as by analysis. Due to space limitations, we limit the exposition to the—in terms of evicting cache blocks—smallest and the largest preempting tasks `edn` and `pilot`. Note the logarithmic scale on the vertical axis. We employ a fairly high associativity of 8, as higher associativities increase the flexibility of the partitioning scheme. Cases where partitioning does not outperform Selfish-LRU for *any* n, or only for $n > 10000$, are indicated by > 10000 in the figures. Note that the static analysis results are not necessarily upper bounds to the respective simulation results. Analysis imprecisions may influence the results in both directions: the larger the overestimation of the CRPD, the smaller n; on the other hand, varying overestimations of the execution times for different associativities may result in a larger n.

In some scenarios, partitioning outperforms Selfish-LRU even for $n = 1$. This happens, if the cache can be partitioned without incurring any additional intra-task interference in task A or in task B, i.e., if both A's and B's reuse is limited to young data. In such cases, the CRPD may still be non-zero if the preempting task B accesses a lot of data in each cache set.

As an example consider streaming applications, which may access large amounts of data without any reuse.

There are also scenarios in which Selfish-LRU outperforms partitioning for *any* n. This is the case if the preempting task B profits greatly from each additional cache way, while the CRPD is low. In our setting, each task is assigned at least one way, thus limiting the preempting task's share to 7 cache ways. As an example, consider the case that `cruisecontrol` is preempted in Figure 9 and 10.

In many cases in between the two extremes, n is greater than 100. As a ratio of more than 100 between the periods of two tasks occurs only rarely in realistic systems, Selfish-LRU is preferable to partitioning in such cases. Consider, e.g., `stopwatch` and `robodog` in Figure 10.

Interestingly, the values determined using simulation and static analysis differ greatly, and little correlation is visible. Overall, the simulation results favor Selfish-LRU over partitioning more strongly than the static analysis results, indicating that CRPD bounds as they combine approximations of both the preempting and the preempted task are still less precise than WCET bounds.

D. Summary

We conclude that Selfish-LRU outperforms LRU in scenarios where some but not all useful cache contents get evicted. This has been confirmed by both simulation and static analysis.

Concerning partitioning we conclude that even in scenarios consisting of two tasks only, Selfish-LRU is often preferable to partitioning. For larger task sets performance achieved with partitioning will suffer even more. As Selfish-LRU and partitioning have advantages in quite different scenarios, it might be valuable to combine both approaches. It can for instance be useful to limit the number of cache ways a particular task may use to bound the damage a preempting task may incur on other tasks. E.g. in case of a streaming application that would not benefit from additional cache ways anyway.

VII. Conclusions and Future Work

We have introduced Selfish-LRU, a variant of LRU, to improve both *performance* and *predictability* in multitasking systems. While Selfish-LRU behaves the same as LRU in non-preemptive scenarios, it reduces the number of cache misses in preemptive scenarios by preventing *reordering context-switch misses*. Our experimental evaluation demonstrates both improved *observed* and *predicted* performance. While we have introduced and analyzed Selfish-LRU in the context of embedded real-time systems, it may be of interest in other domains as well: Liu et al. [3] have shown that reordering misses constitute a significant share of all context-switch misses in general-purpose systems.

In many cases, CRPD bounds computed by static analysis are close to the CRPD observed in simulations. In some cases, however, CRPD bounds appear to be very imprecise both for LRU and Selfish-LRU. It may be worthwhile to further investigate the sources of overestimation in such cases. To this end, we plan to extend the simulator to record information about the actual numbers of useful and evicting cache blocks, and to force preemptions at particular program points rather than points in time.

Acknowledgements

We thank Sebastian Hack for helpful remarks about this work. This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS), by the Saarbrücken Graduate School of Computer Science, which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments, by an Intel Early Career Faculty Award, and by COST Action IC1202: Timing Analysis On Code-Level (TACLe).

References

[1] J. Schneider, "Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems," in *RTSS*, 2000, pp. 195–204. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/REAL.2000.896009

[2] S. Altmeyer, R. I. Davis, and C. Maiza, "Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems," in *RTSS*, 2011, pp. 261–271. [Online]. Available: http://dx.doi.org/10.1109/RTSS.2011.31

[3] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker, "Characterizing and modeling the behavior of context switch misses," in *PACT*. New York, NY, USA: ACM, 2008, pp. 91–101. [Online]. Available: http://dx.doi.org/10.1145/1454115.1454130

[4] C. Burguière, J. Reineke, and S. Altmeyer, "Cache-related preemption delay computation for set-associative caches: Pitfalls and solutions," in *WCET*, 2009. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2009/2285/pdf/Burguiere.2285.pdf

[5] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999. [Online]. Available: http://dx.doi.org/10.1023/A%3A1008186323068

[6] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007. [Online]. Available: http://rw4.cs.uni-saarland.de/~grund/papers/rts07-predictability.pdf

[7] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970. [Online]. Available: http://dx.doi.org/10.1147/sj.92.0078

[8] T. Sudarshan, R. A. Mir, and S. Vijayalakshmi, "Highly efficient LRU implementations for high associativity cache memory," in *Conference on Advanced Computing and Communications*, Ahemdabad, Gujarat, India, 2004, pp. 87–95. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.7233&rep=rep1&type=pdf

[9] B. Vinnakota, "A new circuit for maximum value determination," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 41, no. 12, pp. 929–930, 1994. [Online]. Available: http://dx.doi.org/10.1109/81.340863

[10] C.-G. Lee, J. Hahn, S. L. Min *et al.*, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," in *RTSS*. IEEE, 1996, p. 264. [Online]. Available: http://dl.acm.org/citation.cfm?id=827268.828946

[11] H. S. Negi, T. Mitra, and A. Roychoudhury, "Accurate estimation of cache-related preemption delay," in *CODES+ISSS*. ACM, 2003. [Online]. Available: http://dx.doi.org/10.1145/944645.944698

[12] Y. Tan and V. J. Mooney, "Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems," in *Workshop on Software and Compilers for Embedded Systems*, 2004, pp. 182–199. [Online]. Available: http://codesign.ece.gatech.edu/publications/ydtan/paper/sc04_Yudong_Tan.pdf

[13] J. Staschulat and R. Ernst, "Scalable precision cache analysis for real-time software," *Trans. on Embedded Computing Systems*, vol. 6, no. 4, p. 25, 2007. [Online]. Available: http://dx.doi.org/10.1145/1274858.1274863

[14] S. Altmeyer and C. Burguière, "A new notion of useful cache block to improve the bounds of cache-related preemption delay," in *ECRTS*. IEEE, 2009, pp. 109–118. [Online]. Available: http://dx.doi.org/10.1109/ECRTS.2009.21

[15] H. Tomiyama and N. D. Dutt, "Program path analysis to bound cache-related preemption delay in preemptive real-time systems," in *CODES*. ACM, 2000. [Online]. Available: http://dx.doi.org/10.1145/334012.334025

[16] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: Tightening the CRPD bound for set-associative caches," in *LCTES*. New York, NY, USA: ACM, 2010, pp. 153–162. [Online]. Available: http://rw4.cs.uni-saarland.de/~reineke/publications/ResilienceAnalysisLCTES10.pdf

[17] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, July 2009. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2009.2013287

[18] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis – definition and challenges," in *6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, December 2013. [Online]. Available: http://embedded.cs.uni-saarland.de/publications/TowardsCompositionality.pdf

[19] C.-G. Lee, J. Hahn, Y.-M. Seo *et al.*, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *Trans. on Computers*, vol. 47, no. 6, pp. 700–713, 1998. [Online]. Available: http://dx.doi.org/10.1109/12.689649

[20] D. Grund, J. Reineke, and G. Gebhard, "Branch target buffers: WCET analysis framework and timing predictability," *Journal of Systems Architecture*, vol. 57, no. 6, pp. 625–637, 2011. [Online]. Available: http://rw4.cs.uni-saarland.de/~grund/papers/jsa10-BTBs.pdf

[21] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg, "Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing," in *CASES*, 2005, pp. 213–224. [Online]. Available: http://dx.doi.org/10.1145/1086297.1086326

[22] S. Edwards and E. Lee, "The case for the precision timed (PRET) machine," in *DAC*. San Diego, CA, USA: IEEE, June 2007, pp. 264–265. [Online]. Available: http://dx.doi.org/10.1145/1278480.1278545

[23] J. Barre, C. Rochange, and P. Sainrat, "A predictable simultaneous multithreading scheme for hard real-time," in *ARCS*, 2008, pp. 161–172. [Online]. Available: http://dl.acm.org/citation.cfm?id=1787770.1787789

[24] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer, "Exploiting spare resources of in-order SMT processors executing hard real-time threads," in *ICCD*, 2008, pp. 371–376. [Online]. Available: http://dx.doi.org/10.1109/ICCD.2008.4751887

[25] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET microarchitecture implementation with repeatable timing and competitive performance," in *ICCD*, September 2012. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/919/ptarm-iccd-2012-accepted-version.pdf

[26] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *CODES+ISSS*. ACM, 2007, pp. 251–256. [Online]. Available: http://dx.doi.org/10.1145/1289816.1289877

[27] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: bank privatization for predictability and temporal isolation," in *CODES+ISSS*, 2011, pp. 99–108. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/851/controller.pdf

[28] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *DATE*, 2002, pp. 409–419. [Online]. Available: http://dl.acm.org/citation.cfm?id=882452.874376

[29] J. Whitham and N. C. Audsley, "Explicit reservation of local memory in a predictable, preemptive multitasking real-time system," in *RTAS*. Washington, DC, USA: IEEE, 2012, pp. 3–12. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2012.19

[30] F. Mueller, "Compiler support for software-based cache partitioning," *SIGPLAN Not.*, vol. 30, no. 11, pp. 125–133, 1995. [Online]. Available: http://dx.doi.org/10.1145/216633.216677

[31] S. Plazar, P. Lokuciejewski, and P. Marwedel, "WCET-aware software based cache partitioning for multi-task real-time systems," in *WCET*, 2009. [Online]. Available: https://ls12-www.cs.tu-dortmund.de/daes/media/documents/publications/downloads/2009-wcet.pdf

[32] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *RTSS*, 2002, pp. 114–124. [Online]. Available: http://dl.acm.org/citation.cfm?id=827272.829141

[33] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *Trans. on Embedded Computing Systems*, vol. 7, no. 1, pp. 4:1–4:38, 2007. [Online]. Available: http://dx.doi.org/10.1145/1324969.1324973

[34] Y. Tan and V. J. Mooney, "WCRT analysis for a uniprocessor with a unified prioritized cache," in *LCTES*. New York, NY, USA: ACM, 2005, pp. 175–182. [Online]. Available: http://dx.doi.org/10.1145/1065910.1065935

APPENDIX

A. Potential Variants and Refinements of Selfish-LRU

For simplicity, our description of Selfish-LRU does not distinguish between reads and writes. However, a variant of Selfish-LRU following a no-write allocate write policy is equally feasible.

Selfish-LRU, as described in Section II, prioritizes memory blocks of the active task over those of other tasks. One might introduce different priorities to distinguish different inactive tasks. Contrary to the first intuition, we believe that it might be valuable to assign higher priorities to memory blocks of lower priorities tasks. However, this remains to be evaluated empirically.

Memory instructions that force memory accesses to bypass the cache can be used to improve cache performance. When such instructions are available, the compiler needs to decide where to bypass the cache. Within an individual task the costs and benefits of an additional memory access in terms of cache hits and misses can be approximated statically. In preemptive scenarios, however, this previously required precise knowledge of the set of coscheduled tasks. With Selfish-LRU, an additional memory access in the preempted task may yield at most one additional cache miss in other tasks. This enables the compiler to make an informed decision about bypassing a particular memory access, in a multitasking scenario without knowledge of the schedule or the coscheduled tasks.

B. Selfish-LRU is a Stack Algorithm: Proof Sketch

As observed by Mattson et al. [7], a sufficient condition for a policy to be a *stack algorithm* is that at any point in time, there is a total ordering on all previously referenced memory blocks that is independent of the cache size, such that, upon a miss, the policy replaces the greatest (according to the total ordering) memory block among the current cache contents.

In Selfish-LRU, this total ordering is defined as follows. Let t_a be the active task, and for any $x \in B$, let $age(x)$ denote the number of memory accesses since the last access to x. Then:

$$(a_1, t_1) < (a_2, t_2) :\Leftrightarrow (t_1 = t_a \wedge t_2 \neq t_a) \vee (t_1 = t_2 \wedge age(a_1) < age(a_2)).$$

This defines a total ordering on all previously referenced blocks, as for all a, b that have been referenced before $age(a) \neq age(b)$.