Work-In-Progress: Reducing Cache Conflicts via Interrupts and BUNDLE Scheduling

Corey Tessler
Wayne State University
corey.tessler@wayne.edu

Gedare Bloom
Howard University
gedare@scs.howard.edu

Nathan Fisher
Wayne State University
fishern@wayne.edu

***Abstract*—In "BUNDLE: Real-Time Multi-Threaded Scheduling to Reduce Cache Contention" Tessler and Fisher present a positive perspective of instruction caches for hard real-time multi-threaded tasks. The thread-aware scheduling algorithm limits the execution of threads to sets of instructions that cannot result in cache conflicts. Identification of these sets result in conflict free regions which are used to identify scheduling groups called bundles in the BUNDLE scheduling algorithm. Placement of a thread in a particular bundle depends on, what the authors call, "anticipating execution". However, they do not define a complete mechanism to anticipate execution.**

In this work, we propose a method to anticipate execution that modifies cache hardware and introduces a new interrupt raised prior to a cache conflict. This new interrupt is combined with (a slightly modified version of) the BUNDLE scheduling algorithm. The intent is to implement these hardware modifications for ARM on the gem5 simulator with the scheduling algorithm integrated into the RTEMS operating system. The hope is this work serves as further motivation to bring the positive perspective of caches to physical processors and operating systems.

Keywords*—*Scheduling algorithms, Cache Memory, Multi-threading, Static Analysis

I. Introduction

In BUNDLE [1] a positive perspective on the shared resource of instruction caches is presented in the context of hard real-time multi-threaded tasks. Previous analytical techniques for worst case execution time [2, 3], cache related preemption delay [4]–[6], scheduling [7], and schedulability analysis [8]–[10] operated independently. A unified approach is proposed, one that shares information between all components.

The setting is limited to a single multi-threaded task, releasing m threads per job. Within a task, there may be multiple instructions that serve as the entry point of a thread. An entry instruction and the set of instructions reachable from it are referred to as a *ribbon*.

A task τ_i belongs to the set of all tasks τ and contains several ribbons $\rho_i, \rho_{i+1}, ...$ in the set of all ribbons $\boldsymbol{\rho}$. Each task releases jobs, denoted τ_i^j where j is the index of the job. Exactly m threads are released with each job. A thread begins execution with the entry instruction of a ribbon ρ_i and is denoted ρ_i^k where $k \in \{1..m\}$ is the index of the thread.

This research has been supported in part by the US National Science Foundation (CNS Grant Nos. 1205338, 1618185 and 1646317)

c

A thread-aware scheduling algorithm BUNDLE selects which thread may run at any time instant. Thread selection depends on the bundle which a thread belongs to at any time. Only threads of the active bundle are allowed to execute, and only one bundle is allowed to be active. A thread moves from one bundle to another by *attempting* to execute an instruction that would result in a conflict.

To identify at runtime an attempted instruction execution, a short proposal for a hardware mechanism was proposed alongside BUNDLE. However, the brief description is insufficient for a physical implementation. This work furthers support for BUNDLE by proposing a CONFLICT interrupt that meets the requirements of the scheduling algorithm. The proposed modifications to cache memory described in Section II. The cache modifications are integrated into BUNDLE's scheduling algorithm in Section III. Since this work is currently in progress, we summarize our current results and describe the remaining effort in Section IV.

II. Cache Modifications

Current cache and memory architectures are insufficient for BUNDLE's scheduling algorithm. The scheduling algorithm depends on static analysis to determine which instructions can lead to cache conflicts. To be reliable, static analysis depends upon BUNDLE to block threads when they attempt to execute conflicting instructions. At the time of this work, no mechanism is known that can detect a thread attempting to execute a conflicting instruction. This is due (in part) to the modular design of caches, operating as passageways for data from main memory to the CPU without burdening the processor with cache management. The opaque nature of modular design has precluded the inspection of cache contents.

To allow BUNDLE to respond to the intended execution of conflicting instructions, modifications to cache memory are proposed. The focus of the changes are a set of new flag bits named *conflict bits* or xbits. Each cache block is extended by the set of conflict bits, which have their value set by a new XMASK register. These bits act as a filter to raise *potential* CONFLICT interrupts to the processor. A description of the novel interrupt mechanism relies on a working model of cache memory. What follows is a brief description of set associative caches [11], which is then modified to include conflict bits.

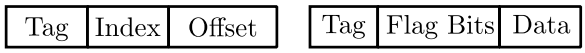
A. Cache Background

An n-way set associative cache places n cache blocks within the same cache set. A cache block holds copies of one or more words addressable in main memory.

978-1-5090-5269-1/17 $31.00 © 2017 IEEE

An address in main memory is referred to as an effective address and is divided into three parts: a tag, index, and offset as depicted in Figure 1a. The index identifies the cache set of the block, and the offset specifies the location within the block to find the data.

Only sets have addresses within a cache. To determine if the data of particular address is present the cache block must also store the tag of the effective address—as shown in Figure 1b. In addition to the tag, each block carries flag bits to indicate if the entry is valid or dirty.



(a) Effective Address (b) Cache Block

Fig. 1: Addresses and Cache Blocks

The relationship between data from main memory and the corresponding location in cache memory is illustrated in Figure 2. For simplicity, main and cache memory are presented as a sequence of cache block sized regions which also allows the offsets to be omitted.

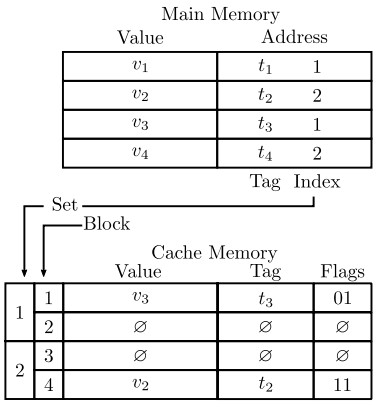


Fig. 2: Main Memory to Cache Memory

B. Extending Cache Flags with xbits

Figure 3 highlights the extension proposed in this paper. Each cache block is extended by a few bits to store conflict bits (xbits). The number of bits required for the xbits field depends on the number of ribbons and is equal to $\max(1, log_2|\boldsymbol{\rho}|)$. There are no restrictions on the values of conflict bits.

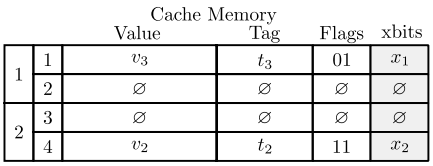


Fig. 3: Conflict Bits

Conflict bits are added to cache blocks as a side effect of loading. A cache load (which we assume occurs on a cache miss) copies the value from main memory and tag from the address to the block. The change, proposed herein, copies the value from our newly-proposed `XMASK` register to the xbits of cache blocks when they are loaded. There is a special `XMASK` value of *clear* and is represented by the symbol ∅. It indicates that the xbits have not been set or is to be ignored.

The contents of the `XMASK` register are under control of the CPU, which can read and write any value to it. The cache is permitted to read, but not write the contents of the `XMASK` register. Similarly, the CPU is not given visibility into the cache to inspect cached values, addresses, or xbits.

In addition to affecting cache loads, the `XMASK` register influences cache flush commands such as the `WBINVD` command found in Intel processors [12]. When a flush and invalidate is requested only those blocks with matching xbits are cleared. If the `XMASK` register has the value ∅ all cache blocks are cleared.

C. Raising a Conflict Interrupt

Conflict bits on cache blocks and the `XMASK` register are input to a new hardware interrupt. When the interrupt is raised, it indicates the instruction being executed would have resulted in a cache miss (and load) if it were allowed to perform the requested memory access. Raising an interrupt for every cache miss would be excessive, to limit the conditions under which the interrupt is raised the `XMASK` register is used as a filter.

Pseudo-code for the proposed hardware interrupt is given in Figure 4. Cache memory is represented by an array C. An index i into C addresses a single cache block. A cache block is a tuple containing a tag t, flag bits f, xbits x, and value v, e.g. $C[i] = (t, f, x, v)$.

The presence of two hardware facilities are required for the conditional load procedure. The first BLOCK(a) determines the proper cache block index of an effective address a. It is the responsibility of BLOCK(a) to select the appropriate cache block index when a is present or absent from the cache. If a is absent from the cache and the cache set a belongs to contains any cached block with index i and xbits $\neq \varnothing$, i must be returned by BLOCK(a). For the purposes of this paper the replacement policy is immaterial. The FETCH(a) procedure delivers from main memory the value at address a.

```
1:  procedure COND_LOAD(a)
2:      i ← BLOCK(a)
3:      x ← C[i] = (t, f, x, v)
4:      if x = XMASK then
5:          raise(CONFLICT)
6:      else
7:          v ← FETCH(a)
8:          C[i] ← (t, f, XMASK, v)
9:      end if
10: end procedure
```

Fig. 4: Conditional Load

The interrupt mechanism is straightforward. When a cache load is *requested* for block i, extract the current xbits value as x ($C[i] = (t, f, x, v)$). If $x \neq$ `XMASK` perform the cache load. If $x =$ `XMASK` do not perform the cache load into $C[i]$ and raise a `CONFLICT` interrupt. From the CPU's perspective, a memory access that results in a `CONFLICT` interrupt is a rejection of the access while maintaining the current state of the cache.

A CONFLICT interrupt is precise, instructions present in the pipeline that precede the conflicting instruction in program order are committed and those succeeding are removed. The conflicting instruction is removed from the pipeline without a lasting effect. After handling the CONFLICT interrupt at the CPU an interrupt service routine (ISR) is invoked, allowing the scheduler to respond.

III. Scheduler Integration

An interrupt mechanism is proposed in BUNDLE [1], but lacks a definition or description of integration with the scheduling algorithm BUNDLE. In this section, a possible definition and integration are given along with justification for their construction. To do so, several portions of BUNDLE [1] are repeated and summarized to clarify the interaction with the new hardware mechanism.

Under BUNDLE threads are scheduled based on conflict free regions. A conflict free region is a sub-graph of the control flow graph of a task. A control flow graph [13] is a representation of the instructions of a task that models the valid paths of execution a thread may take. Analysis of the control flow graph identifies sub-graphs with the properties 1.) no two instructions map to the same cache block (set) 2.) each sub-graph contains a single entry point. These sub-graphs are named conflict free regions. Figure 5 illustrates the relationship between a control flow graph (CFG) and conflict free regions (CFRs).

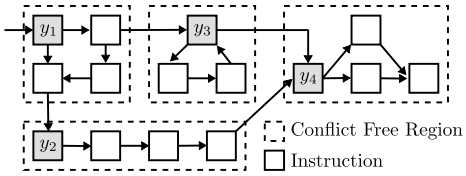


Fig. 5: CFG and CFRs

Vertexes highlighted in Figure 5 are the entry instructions of conflict free regions, and they uniquely identify each region by their address. Using the entry instructions as vertexes and maintaining connectivity from the CFG a new graph is generated called the conflict free region graph abbreviated (CFRG). Figure 6 is the CFRG corresponding to the CFG in Figure 5.

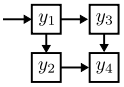


Fig. 6: CFRG

The entry instructions of CFRs are used to make scheduling decisions by BUNDLE. Each address $y_i \in Y$ identifies a bundle, which threads belong to. Recall that one bundle is active, only threads of the active bundle are permitted to execute. At job release, all threads of the same ribbon are awaiting to execute the same instruction and are placed within the same bundle. While executing in the active bundle if a thread attempts to execute an instruction outside of the associated CFR the thread is placed in a new bundle and suspended.

Pseudocode of the scheduling algorithm is given in Figure 7. A small modification is required to the BUNDLE procedure. At line 11 when the condition is true a new bundle is being selected as active. As a new bundle is being selected, the cache must be flushed to clear any errant xbits values. Line 15 was added to avoid spurious interrupts.

```
1:  R                                              ▷ Set of Threads
2:  Y                ▷ Set of Conflict Free Region Entry Points
3:  procedure BUNDLE
4:      A ← R                                      ▷ Active Bundle
5:      B ← ∅          ▷ Inactive Bundles (Blocked Threads)
6:      while true do
7:          ρ^i ← a, a ∈ A                         ▷ Select a thread
8:          RUN(ρ^i) until ρ^i's next instruction is y ∈ Y
9:          B[y] ← B[y] ∪ ρ^i
10:         A ← A ∖ ρ^i
11:         if |∪_{y∈Y} B[y]| = |R| then
12:             Select z ∈ Y, |B[z]| ≠ 0
13:             A ← B[z]
14:             B[z] ← ∅
15:             CFLUSH()
16:         end if
17:     end while
18: end procedure
```

Fig. 7: BUNDLE [1] Scheduling Algorithm

Line 8 of the pseudocode was previously undefined and is the focus of this work. Line 8 requests a thread ρ^i be run until the next instruction executed is in the set Y indicating the thread is leaving the current conflict free region. It is the cache modification and interrupt mechanism defined in the previous section that enables the desired behavior.

Presented as pseudocode in Figure 9, the RUN procedure relies on two products of static analysis: the existing *next inter-thread cache conflicts* ($P(a)$) and the new *conflictors* ($P'(a)$). Next inter-thread cache conflicts from an entry instruction $a \in Y$ are a set of entry points in subsequent conflict free regions $A \in Y$. The new conflictors function maps an entry instruction $a \in Y$ to the set of addresses B that conflicted with $P(a)$. These are the instructions that conflicted with the next inter-thread cache conflicts. All instructions of $P'(a) = B$ are found within the conflict free region identified by a.

Figure 8 illustrates the relationship between inter-thread conflicts and conflictors. For CFR a_1 the set of next inter-thread conflicts (which is a superset of the intra-thread conflicts) is $P(a_1) = \{a_2, a_3\}$. The instructions conflict with $\{b_1, b_2\}$ since $\{a_2, a_3\}$ use the same cache blocks respectively. Therefor, $P'(a) = \{b_1, b_2\}$.

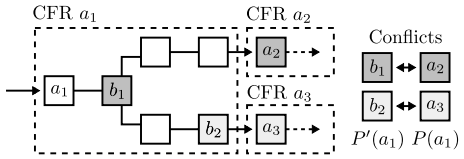


Fig. 8: Conflicts

In addition, the following facilities provided by the operating system and architecture are utilized. The functions SAVE_CTX and RESTORE_CTX save and restore the context of a thread ρ. A thread has a current program counter available as ρ.pc. To flush and invalidate the cache CFLUSH clears the cache block values, tags, and xbits which match the current value of the XMASK register—or all blocks if the value is $\varnothing$.

```
1: procedure RUN(ρ)
2:     B ← P′(ρ.pc)          ▷ Instructions that will conflict
3:     XMASK ← ρ ▷ Set the XMASK register for the cache
4:     for all b ∈ B do
5:         PREFETCH(b)
6:     end for               ▷ Cache blocks ∀b ∈ B xbits are set
7:     RESTORE_CTX(ρ)                ▷ Begin executing ρ
8:                   ▷ ρ executes until completion or interrupt
9:     SAVE_CTX(ρ)
10: end procedure
```

Fig. 9: RUN Procedure

A PREFETCH operation is required, one that performs a cache load for the specified block. The load copies the value from main memory, stores the proper tag, and copies the XMASK value to the xbits of the cache block. Some ARM processors [14] provide a prefetch instruction (PRFM) which could be used in place of PREFETCH in the pseudocode (presuming the xbits are set according to the current XMASK value). No other instruction pre-fetching is permitted, if provided by the processor it must be disabled.

There are three conceptual sections to the RUN algorithm, preparation, execution, and resumption. Preparation caches the conflictor values which sets the xbits of each cache block for instructions that will conflict with the entry instructions of subsequent conflict free regions. Execution restores the context of the thread before executing it by jumping to the program counter.

During execution, an instruction a will fall into one of three categories:

1) Cache miss with xbits value of $\varnothing$.
2) Cache miss with xbits value of ρ.
3) Cache hit with xbits value of ρ.

Case 1 corresponds to an instruction being executed for the first time, having never been cached. Case 3 is the result of an instruction having been cached previously as part of preparation or execution by a thread within the same conflict free region. Case 2 indicates that a thread is attempting to execute an instruction outside of the current conflict free region, which raises a CONFLICT interrupt. The ISR for the interrupt stores the conflicting instruction in ρ.pc. When the ISR exits it returns to line 8 of the RUN procedure. Afterwards, the thread context is saved (SAVE_CTX) before returning to BUNDLE's main scheduling loop.

It is by necessity that the cache is flushed when selecting a new active bundle. Earlier cache loads may have populated cache block xbits with the current XMASK value. Without a flush, stale contents could result in spurious CONFLICT interrupts. These additional cache flushes do not interfere with the timing analysis of conflict free regions presented in [1], but they do add a new context switch cost when selecting a new active bundle—one flush per bundle switch.

IV. Conclusion and Ongoing Effort

The proposed CONFLICT interrupt mechanism, which prevents cache conflicts based on a XMASK filter, allows the BUNDLE scheduling algorithm to be fully implemented. Additionally, the proposed CONFLICT interrupt is compatible with the ongoing efforts to expand BUNDLE from a single to multi-task environment.

However, this work is focused on bringing the interrupt mechanism and scheduling algorithm closer to a physical implementation. The ongoing effort is to implement the hardware components in the gem5 [15] simulator for an ARM processor, along with implementing the BUNDLE scheduling algorithm on top of RTEMS [16].

References

[1] C. Tessler and N. Fisher, "BUNDLE: Real-Time Multi-Threaded Scheduling to Reduce Cache Contention," *IEEE Real-Time Systems Symposium*, 2016.

[2] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding worst-case instruction cache performance," *Real-Time Systems Symposium, 1994., Proceedings.*, pp. 172–181, Dec 1994.

[3] F. Mueller, "Static cache simulation and its applications," Ph.D. dissertation, Florida State University, 1995.

[4] S. Altmeyer and C. Maiza Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, Aug. 2011.

[5] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, Jun. 1998.

[6] H. S. Negi, T. Mitra, and A. Roychoudhury, "Accurate estimation of cache-related preemption delay," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03. New York, NY, USA: ACM, 2003, pp. 201–206.

[7] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[8] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "Proartis: Probabilistically analyzable real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, May 2013.

[9] J. M. Calandrino and J. H. Anderson, "Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study," in *2008 Euromicro Conference on Real-Time Systems*, July 2008, pp. 299–308.

[10] A. Alhammad and R. Pellizzoni, "Time-predictable execution of multithreaded applications on multicore systems," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2014, pp. 1–6.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[12] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2*, Intel Corporation, 2016. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[13] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970.

[14] ARM, *ARM Cortex-A57 MPCore Processor Technical Reference Manual*, ARM, 2016. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488h/DDI0488H_cortex_a57_mpcore_trm.pdf

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[16] G. Bloom and J. Sherrill, "Scheduling and thread management with rtems," *SIGBED Rev.*, vol. 11, no. 1, pp. 20–25, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2597457.2597459