# Scheduling Multi-Threaded Tasks to Reduce Intra-Task Cache Contention

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

Nathan Fisher
Wayne State University
fishern@wayne.edu

***Abstract*—Research on hard real-time systems and their models has predominately focused upon single-threaded tasks. When multi-threaded tasks are introduced to the classical real-time model the individual threads are treated as distinct tasks, one for each thread. These artificial tasks share the deadline, period, and worst case execution time of their parent task. In the presence of instruction and data caches this model is overly pessimistic, failing to account for the execution time benefit of cache hits when multiple threads of execution share a memory address space.**

**This work takes a new perspective on instruction caches. Treating the cache as a benefit to schedulability for a single task with $m$ threads. To realize the "inter-thread cache benefit" a new scheduling algorithm and accompanying worst-case execution time (WCET) calculation method are proposed. The scheduling algorithm permits threads to execute across conflict free regions, and blocks those threads that would create an unnecessary cache conflict. The WCET bound is determined for the entire set of $m$ threads, rather than treating each thread as a distinct task. Both the scheduler and WCET method rely on the calculation of conflict free regions which are found by a static analysis method that relies on no external information from the system designer.**

**By virtue of this perspective the system's total execution execution time is reduced and is reflected in a tighter WCET bound compared to the techniques applied to the classical model. Obtaining this tighter bound requires the integration of three typically independent areas: WCET, schedulability, and cache-related preemption delay analysis.**

## I. Introduction

In the classical model of real-time systems, shared resources are often considered detractors to schedulability analysis and exclusively increase worst-case execution times (WCETs). Cache memory is one such shared resource viewed from this exclusively negative perspective. It is a natural perspective, derived from a preempting task invalidating cache lines, thus extending a preempted task's execution time.

For example in the classical periodic task model [1] it is implied that a task is a single thread of execution. These models lack a representation for tasks with multiple threads. To apply WCET and schedulablility techniques developed for the classical models, a task that executes multiple threads is treated as several duplicate tasks with a single thread of execution. Any task that releases a job with $m$ threads will be converted to $m$ tasks each releasing one job.

Under such models, tasks are assumed to be in competition for cache space. The inclusion of threads, which are converted to tasks, only amplifies the negative affect. However, threads are not always in competition with other threads, but in fact can mutually benefit from reusing the same resources. Threads of the same task share a memory space (also referred to as an address space). A cache miss during the execution of one thread can place values into the cache that produce a cache hit for a second thread. These unexpected cache hits reduce the execution time of the second thread and the system overall. This speed up is called the **inter-thread cache benefit**. While other researchers have developed techniques for limiting the impact of caches [2] [3] [4], we are unaware of any existing analysis technique that accounts for the benefit of caches between threads or tasks.

The purpose of this work is to illustrate the potential inter-thread cache benefit for instruction caches, and to argue for a new task model and schedulability analysis technique. Current approaches to Worst Case Execution Time (WCET), Cache-Related Preemption Delay (CRPD), and schedulability analyis typically operate independently. Accounting for the inter-thread cache benefit requires a unified approach, integrating all of these disciplines.

As a first step towards a complete approach this work is limited to a single task executing multiple identical threads. Preemptions between threads incur no time penalty. The main work-in-progress efforts described herein are the development of a new thread based scheduling algorithm called `BUNDLE`, and a method for calculating the WCET bound of $m$ threads scheduled by `BUNDLE`. By the nature of the WCET calculation, the bound permits any number of preemptions between threads. When $m$ is greater than one, the bound is guaranteed to be less than any WCET method using the competitive perspective from the classical model.

## II. Motivating Example

An example in two parts provides the motivation for this work. This example illustrates the inter-thread cache benefit and highlights the pessimism in existing WCET and CRPD approaches. As noted, the classical model provides no representation of a threads distinct from a job. To aid the example the classical model is augmented with two new concepts: threads and ribbons.

Customarily, the term "thread" may refer to the execution of a sub-job, or it may refer to the subset of instructions reachable from an entry point. To clarify the distinction the

This research has been supported in part by an NSF CAREER Grant (No. CNS-0953585), an NSF CRI grant (No. CNS-1205338), and a grant from Wayne State University's Office of Vice President of Research.

term *ribbon* is introduced and refers to the subset of instructions reachable from a single entry point. A *thread* will refer exclusively to one instance of execution i.e., an instantiation of a ribbon.

The computational setting is a single processor with an instruction cache of $l$ lines. The cache is write-through and direct-mapped, assigning exactly one cache line to a memory address. To simplify the presentation, every instruction completes in $\mathbb{I}$ time units. If an instruction is absent from the cache before execution, its completion will be delayed by the Block Reload Time (BRT): $\mathbb{B}$. Executing an instruction out of the cache (i.e. a hit) takes $\mathbb{I}$ time, while caching and executing a miss takes $(\mathbb{I} + \mathbb{B})$.

For periodic and sporatic tasks, the classical model accumulates the $n$ tasks in the set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task is characterized by a tuple of minimum inter-arrival time, relative deadline, and worst case execution time: $\tau_i = (p_i, d_i, c_i)$. Each $c_i$ value is an upper bound on the amount of time one job of $\tau_i$ will take to complete if the job executes without preemption.

To more closely represent the execution of threads within jobs the model must be modified. With that purpose, each task is represented by a tuple of minimum inter-arrival time, relative deadline, and initial ribbon: $\tau_i = (p_i, d_i, r_i)$. A ribbon $r_i$ is identified by a starting instruction within the object of a task, it includes all reachable instructions until an exit point. The set of $m$ ribbons from all tasks is named $R = \{r_1, r_2, ..., r_m\}$, where $|R| \geq |\tau|$. Every ribbon has an associated worst-case execution time: $r_j = c(r_j)$.

TABLE I. SUMMARY OF MODEL PARAMETERS

| Tasks | Task | Ribbons | Ribbon |
|---|---|---|---|
| $\tau_i \in \tau$ | $\tau_i = (p_i, d_i, r_i)$ | $r_j \in R$ | $r_j = c(r_j)$ |

| Cache Lines | Instruction Time | BRT |
|---|---|---|
| $l$ | $\mathbb{I}$ | $\mathbb{B}$ |

### A. Example Part I: 1 Linear Ribbon, 2 Threads

To demonstrate the inter-thread cache benefit, consider the following task set of a single task $\tau = \{\tau_1\}$. The task has only one ribbon, the initial ribbon $r_1$ which readies two threads, $r_1^1$ and $r_1^2$ for every job release. For simplicity, $r_1$ contains no loops or branches.

TABLE II. EXAMPLE MODEL PARAMETERS

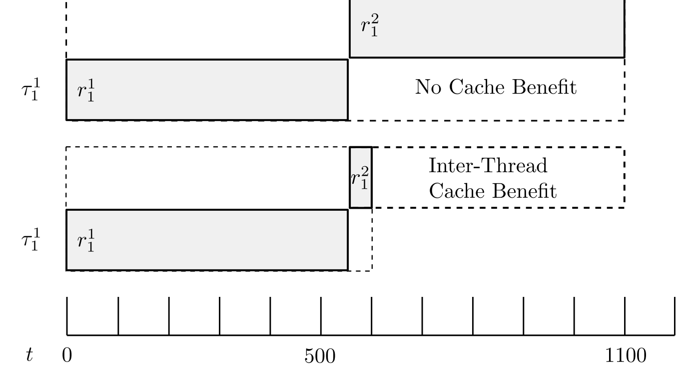
| Tasks | Task | Ribbons | Ribbon Length |
|---|---|---|---|
| $\tau = \{\tau_1\}$ | $\tau_1 = (p_1, d_1, r_1)$ | $R = \{r_1\}$ | $\|r_1\| = 50$ |

| Cache Lines | Instruction Time | BRT |
|---|---|---|
| $l = 200$ | $\mathbb{I} = 1$ | $\mathbb{B} = 10$ |

For every job release, consider a scheduling algorithm that runs one thread of $r_1$ to completion before permitting the second thread to begin execution. The ribbon $r_1$ is 50 instructions long, without loops or branches all 50 instructions miss the cache and each take $(\mathbb{I} + \mathbb{B})$ = 11 cycles to complete for a total of 550 cycles per thread. An analysis that does not consider the benefit of the cache between threads of $r_1$ will reserve 1100 cycles for the combined execution of $r_1^1$ and $r_1^2$.

Since $\tau_1$ (and $r_1$) contain less than $l$ uniquely addressed instructions, every instruction of $\tau_1$ maps to a distinct address in the cache. Assuming no cache flushes are permitted during the execution of $r_1^1$, the execution time of 1100 cycles is reduced to 600 cycles by the inter-thread cache benefit. Figure 1 illustrates the reduction.

Fig. 1. Inter-Thread Cache Benefit



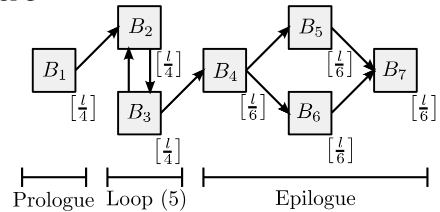

During the execution of $r_1^1$ every instruction is placed in the cache consuming 550 cycles. Since $r_1^1$ and $r_1^2$ share the same address space, all instructions are available in the cache during $r_1^2$'s execution; taking 50 cycles to complete. Also note, in this task system 600 cycles is the worst case execution time for $\tau_1$. Regardless of the thread execution order, one thread will execute an instruction before the other thread caching the instruction value. When the later thread executes an instruction, it will find the value present in the cache.

### B. Example Part II: Existing WCET & CRPD Approaches

To illustrate the benefit of a new approach, the existing approaches to WCET and CRPD analysis are applied to a slightly more complicated ribbon. The advantage will be shown by applying the cache aware methods of Arnold [5] and Mueller's [6] for WCET calculation, and Lee et al.'s [7] CRPD determination. These methods were chosen for illustrative purposes and for their continued use in subsequent works.

The structure of $r_1$ from the previous example is ill suited for meaningful WCET or CRPD analysis. To continue, $r_1$ is modified to include a prologue, loop, and epilogue for a total instruction count of $\frac{5}{4} \cdot l$. Figure 2 gives the control flow graph [8] (CFG) of $r_1$ which connects serial sets of instructions, called basic blocks, by their logical control flow through the ribbon. Below each basic block is a counting term in square brackets listing the number of instructions in the block. The parenthesized value at the bottom of the figure indicates the number iterations the loop will execute.

Fig. 2. $r_1$ CFG



*1) WCET:* Predominantly, WCET analysis that includes cache behavior is limited to a single task, specifically between preemption points [5]. Arnold [5] and Mueller's [6] approach iterates over the control flow graph categorizing instructions as cache must-miss, first-miss, first-hit, and must-hit. Table III lists the result of categorization for each basic block using Arnold's approach.

TABLE III. BASIC BLOCK CATEGORIZATION

| $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ | $B_6$ | $B_7$ |
|---|---|---|---|---|---|---|
| must-miss | first-miss | | must-miss | | | |

Using these categorizations and the loop bound, the worst case execution time of $r_1$ is the sum of the excution times of the prologue, the entry executions of $B_2$ and $B_3$, the repititions of $B_2$ and $B_3$, and the epilogue. Table IV gives the intermediate values, using the model parameters of $\mathbb{B} = 10$, $l = 200$, and $\mathbb{I} = 1$ the total execution time taking into consideration reloads is: $\frac{l(\mathbb{B}+\mathbb{I})}{4} + \frac{2l(\mathbb{B}+\mathbb{I})}{4} + \frac{8l(\mathbb{I})}{4} + \frac{3l(\mathbb{B}+\mathbb{I})}{6} = \frac{l(5\mathbb{B}+13\mathbb{I})}{4} = 3150$

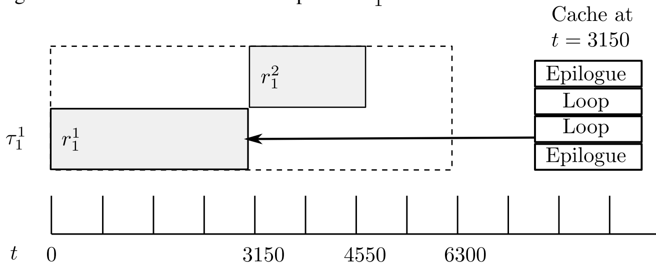
TABLE IV. SEGMENT WCET

| Section | Basic Blocks | WCET |
|---|---|---|
| Prologue | $B_1$ | $\left(\frac{l}{4} \cdot (\mathbb{B} + \mathbb{I})\right)$ |
| Loop Entry | $B_2 + B_3$ | $\left(\frac{l}{4} \cdot 2 \cdot (\mathbb{B} + \mathbb{I})\right)$ |
| Loop Repetition | $(B_2 + B_3) \cdot 4$ (repeats) | $\left(\frac{l}{4} \cdot 2 \cdot 4 \cdot (\mathbb{I})\right)$ |
| Epilogue | $B_4 + (B_5$ or $B_6) + B_7$ | $\left(\frac{l}{6} \cdot 3 \cdot (\mathbb{B} + \mathbb{I})\right)$ |

Using the WCET of 3150 for two threads of $r_1$, where the execution of each thread is considered a distinct task, the total demand for the one task system is 6300. However, this is overly pessimistic. The worst possible schedule for two threads of $r_1$ is the sequential execution of $r_1^1$ followed by $r_1^2$. It is the worst schedule because it inflicts the most cache misses during the execution of $r_1^2$.

The prologue consumes one quarter of the cache, the loop one half, and the epilogue another half. Meaning, after the epilogue executes all of the cache lines of the prologue have been invalidated. Scheduling $r_1^2$ after $r_1^1$ has completed requires $r_1^2$ to load the cache lines of the prologue, and half of the cache lines from the epilogue. With this understanding, the WCET of $r_1^2$ is: $\frac{l(\mathbb{B}+\mathbb{I})}{4} + \frac{5l(\mathbb{I})}{4} + \frac{l(\mathbb{B}+\mathbb{I})}{4} + \frac{l(\mathbb{I})}{4} = \frac{2l(\mathbb{B}+4\mathbb{I})}{4} = 1400$. The total demand for the task system is 4550 which is less than the 6300 calculated from the WCET analysis, demonstrating the pessimism of the Arnold and Mueller approaches. Figure 3 illustrates the worst possible schedule for $r_1^2$ including the cache contents at $t = 3150$, as well as the pessimistic estimate for $\tau_1$'s execution time.

Fig. 3. Worst Schedule with Respect to $r_1^2$




*2) CRPD:* Cache related preemption delay accounts for the execution time extension of one task due to the cache interference of another. A task executing in isolation may store and reuse values from the cache. When preempted, those stored cache values may be invalidated before they are reused. Upon resuming the preempted task must pay the BRT for each invalidated cache block, extending the execution time of the preempted task. This delay is called the Cache Related Preemption Delay (CRPD), and one method for calculating it is the Useful Cache Block (UCB) approach developed by Lee et al. [7].

The UCB approach borrows from the Arnold and Mueller approaches, iterating over the control flow graph to determine if cache blocks are useful or not. A useful cache block is "a cache block that contains a memory block that may be referenced before being replaced by another memory block." – within the same task.

From Figure 2 there are two basic blocks that contribute UCBs to the thread $r_1$: $B_2$ and $B_3$. Applying Lee's method, CRPD of a preemption of $r_1$ is $\frac{2l(\mathbb{B}+\mathbb{I})}{4} = 1100$ the sum of cache lines from $B_2$ and $B_3$. However, this bound is overly pessimistic.

Given the schedule in Figure 3 once the "Loop" instructions are cached they cannot be invalidated. If $r_1$ were to be preempted after the first iteration of the loop the cache lines mapping to the instructions of $B_2$ and $B_3$ would be populated. No other instructions of $r_2$ map to those cache lines, and cannot invalid them. Furthermore, there is no schedule of $r_1^1$ and $r_1^2$ which incurs any CRPD.

Lee's approach to CRPD calculation is known to be an overestimate, there are refinements such as the UCB-ECB [9], UCB-Union, and UCB-Union Multiset [10] approaches. However, the UCB calculation is a component of each of them and the advanced techniques suffer from the same inability to address cache memory as a benefit rather than a detriment. Similarly, the Arnold and Mueller approaches play a role in subsequent WCET methods. For this reason, the fundamental approaches of Lee, Arnold and Mueller were selected for our evaluation of the potential inter-thread cache benefit.

## III. AN OPTIMAL SCHEDULE

The maximum WCET and CRPD bound determined for $r_1$ in the previous section relies on knowledge of the scheduler. A further reduction in WCET and CRPD values is possible by crafting a schedule that considers the cache. Figure 4 is an optimal schedule created by considering the cache contents and thread flow. Each thread will take a different path through the epilogue, $r_1^1$ will take the "high" road, executing the basic blocks in the following order: $\langle B_1, (B_2, B_3)^5, B_4, B_5, B_7\rangle$. The thread $r_1^2$ will take the "low" road, executing: $\langle B_1, (B_2, B_3)^5, B_4, B_6, B_7\rangle$.

Examining the structure and cache usage of $r_1$, the prologue is the problematic section. Upon completion, $r_1$ will remove the cache contents of the prologue. To benefit from the cached values $r_1^2$ must be scheduled to run before the prologue has been invalidated.

The schedule is presented as the series of block executions, the distance between marks on the time axis are not to scale. Above each series of basic blocks is the execution time

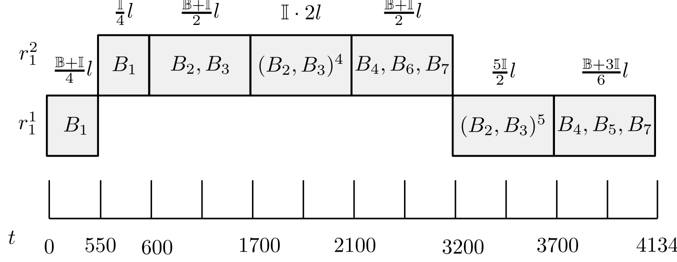


Fig. 4. Optimal Schedule

required to complete the section. By preempting $r_1^1$ after the completion of $B_1$ the total execution time is reduced to $4134$.

## IV. Progress and Remaining Effort

Treating caches solely as detractors in schedulability analysis leads to overly pessimistic WCET bounds for multi-threaded tasks and task systems. This pessimism is further increased by the independent treatment of threads in CRPD analysis. By considering the three disciplines of schedulability, WCET, and CRPD analysis in concert a tighter bound on execution of tasks and task systems can be achieved.

A complete solution that accounts for multiple tasks each with an arbitrary number of ribbons is too ambitious for a first work. By focusing on a single task with one ribbon and $m$ threads, a first useful solution may provide insight for later efforts. Our current research aims include developing a scheduling algorithm (`BUNDLE`), static analysis, and WCET bound for $m$ threads of a single ribbon.

Static analysis provides *conflict free* regions that are used by the scheduling algorithm at run time and off-line timing analysis. A conflict free region is a sub-graph of a ribbon's CFG. It includes a starting instruction and all reachable instructions that cannot create a cache conflict. Conflicts that arise during the execution of a single thread are classified as intra-thread, and those incurred by preemption inter-thread.

At run time, `BUNDLE` uses conflict free regions to group and schedule threads. Intuitively, all threads of the same region are allowed to run in any order and preempt each other arbitrarily. However, when any thread would execute an instruction outside of the region it is blocked until all other threads reach a boundary. After all threads in a region have been blocked, another region is selected for execution.

Determining the WCET of a set of $m$ threads requires the conflict free region boundaries and knowledge of the schedule produced by `BUNDLE`. It also relies upon the static analysis of structures within CFG to establish bounds on the regions. For each conflict free region serial, looping, and branching sub-structures are extracted and help characterize the region with an execution time bound.

Figure 5 illustrates the result of timing characterization for conflict free regions of a ribbon $r$ with CFG $G$. This information, along with the number of threads is passed to the final timing analysis. After finding the longest path through the ribbon, incorporating the behavior of `BUNDLE` over the $m$ threads produces an execution bound for the set of threads. Figure 6 outlines the general approach to timing analysis.

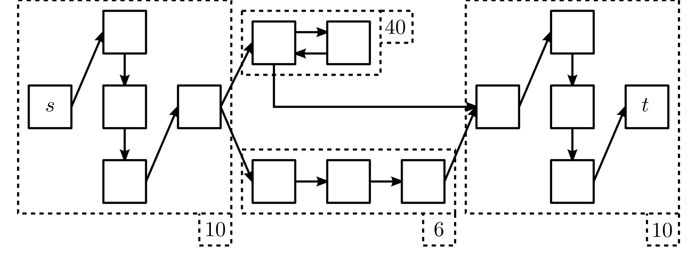


Fig. 5. Sub-Graphs and Bounds for $G$ of $r$

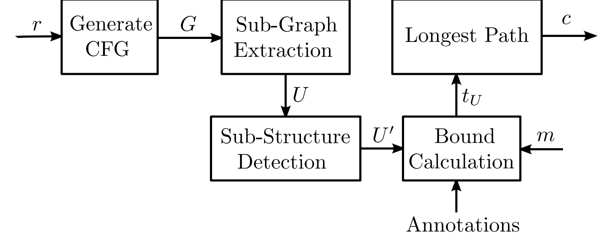


Fig. 6. Timing Analysis Overview

Two significant tasks remain to complete this work: to evaluate and compare. The evaluation will entail more effort due to the lack of tools. Performing an evaluation requires a new memory restricted threading library and CPU simulator with extensible cache configuration. Comparing with related works such as PREM [2] and MultiPREM [3] will emphasize the benefit of combining the three aforementioned disciplines.

## References

[1] 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.

[2] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, April 2011, pp. 269–279.

[3] 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.

[4] M. Schoeberl, W. Puffitsch, and B. Huber, "Towards time-predictable data caches for chip-multiprocessors," in *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, ser. SEUS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 180–191.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.