BUNDLE: Real-Time Multi-Threaded Scheduling to Reduce 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 often share the deadline, period, and worst case execution time of their parent task. In the presence of instruction and data caches this view 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, `BUNDLE`, and accompanying method for calculating the worst-case execution time (WCET) including cache overhead (WCET+O) method are proposed. `BUNDLE` 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+O method rely on the calculation of conflict free regions which are found by a static analysis method of the task object. By virtue of this perspective the system's total execution execution time is reduced and is reflected in a tighter WCET+O 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.

***Keywords*—Scheduling algorithms, Cache Memory, Multi-threading, Static 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. For instance, feasibility analysis for the fork-join model [2] treats WCET separately, where each thread traversing the longest execution path contributes to demand independently.

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. 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**. Other researchers have developed techniques for limiting the impact of caches [3]–[5], or scheduling algorithms to maximize cache-reuse between threads [6]. We are unaware of any existing analysis technique that explicitly quantifies the benefit of caches between threads or tasks.

Concurrent program analysis techniques [7], especially those considering multi-level caches [8] are well poised to realize the benefit. However, these works also treat cache sharing negatively by working to construct the worst-case interleavings of threads that maximize cache overhead. Calandrino's Ph.D. work [9] shifts towards the positive perspective, limiting the "spread" of subtasks (threads), resulting in higher cache hit rates. The improvement is empirically assessed but no analytical method for determining the impact is presented. Our work approaches the problem from the positive perspective, developing scheduling techniques that explicitly avoid these worst-case scenarios and quantify the cache benefit.

The purpose of our paper is to illustrate the potential inter-thread cache benefit for instruction caches, while arguing 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 produce separate values. Accounting for the inter-thread cache benefit requires an approach that integrates the disciplines.

Our long-term research vision is to develop the aforementioned integrated analysis for modern systems comprising several multi-threaded tasks executing upon multi/many-core platforms with potentially multiple levels of cache. However, as far as we know, even the simpler problem of scheduling to formally maximize intra-task cache benefit (and thus reduce a task's execution time) has not yet fully been addressed even for a single multi-threaded task executing on one processor. Thus, in this paper, we take the first step towards our larger vision by solving a necessary first problem: *scheduling the threads of a single multi-threaded task upon a core with a single-level of cache to maximize intra-task cache benefit*. Introducing preemptions between equal-priority threads allows the benefit to be realized and quantified. We believe the solutions developed herein will be fundamental in subsequent research that completes our long-term vision.

The main contributions of this paper are:

1) A positive perspective of instruction caches for threaded processes.
2) A new thread-based scheduling algorithm, BUNDLE.
3) The definition of *conflict free regions* used as a basis for BUNDLE's scheduling decisions.
4) Techniques for identifying *conflict free regions*.
5) A method for calculating the WCET+O bound of m threads scheduled by BUNDLE.
6) An evaluation of BUNDLE's improvement for static timing analysis and runtime (cache and context switch) overhead.

Foremost in this work is the introduction of an integrated and positive perspective of instruction caches when applied to the analytical techniques and execution of multi-threaded tasks. Our new perspective necessarily requires the introduction of a large number of new concepts for this paper. Thus, to facilitate the reader's understanding of these concepts and to fit our paper within the space requirement, we have favored clarity of presentation in the description of our algorithms and theoretical results over efficiency and efficacy. Future research will seek to improve the computational complexity and timing analysis bounds of the initial algorithms presented in this paper. However, we are encouraged that our evaluation shows even for straightforward implementation of our proposed techniques there is still a substantial improvement over the classical analysis approach.

Presentation of the contributions begins with a description of the classical model, the assumptions placed upon it and the limitations of the existing analytical methods in Section II. The section is anchored by a definition of the inter-thread cache benefit aided by examples showing a pessimistic analytical bound from related work and a sub-optimal schedule. Section III describes the BUNDLE scheduling algorithm and introduces conflict free regions, which are formally defined in Section IV along with methodology for their extraction. Section V discusses the analytical techniques for bounding the execution of a task with m threads. Before drawing conclusions and presenting future work two evaluations are given in Section VI: one to evaluate the performance of the integrated approach in obtaining static timing analysis bounds, and another to evaluate the runtime cache and context switch overhead of BUNDLE.

II. Model and Inter-Thread Cache Benefit

From the classical model, a task is a collection of instructions that perform a logical function. A job of a task is a request to execute those instructions. Traditionally, the model assumes each task contains a single starting instruction for all jobs, called the entry point. A scheduling algorithm selects the job to run from those available at any given time.

For example, for periodic [1] tasks, the classical model accumulates the n tasks in the set $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. A 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 τ_i will take to complete if the job executes without preemption. The entry point of a task is implied.

In the classical model, a task contains only one entry point. With the introduction of threads, a task may contain multiple entry points. During the execution of a task, a job may simultaneously release multiple sub-jobs starting from any of the available entry points.

Customarily, the term "thread" may refer to the execution of a sub-job, or it may refer to the instructions reachable from an entry point. To clarify the distinction, the term *ribbon* is introduced and refers to the instructions reachable from a single entry point. A *thread* will refer exclusively to one instance of execution i.e., an instantiation of a ribbon. Analysis of multi-threaded tasks in the classical model relies on each thread being converted to an independent task. Such tasks are referred to as *synthetic tasks*.

To more closely represent the execution of threads within jobs, a modification to the classical model is proposed and summarized in Table I. Tasks are represented by a tuple of minimum inter-arrival time, relative deadline, and initial ribbon: $\tau_i = (p_i, d_i, \rho_i)$. A ribbon ρ_i is identified by a starting instruction within the executable object of a task; it includes all reachable instructions until an exit point. The set of k ribbons from all tasks is named $\rho = \{\rho_1, \rho_2, ..., \rho_k\}$, where $|\rho| \geq |\tau|$.

For a task τ_i, a job is released no earlier than p_i time units since the previous release. Jobs are indexed by their release starting with τ_i^1. Concurrent with the release of a job an initial thread of the ribbon ρ_i is also released. Threads are also indexed by their release starting with ρ_i^1. A thread (initial or otherwise) may release additional threads during its execution. As a shorthand, when referring to an arbitrary thread j of a ribbon ρ_i, the notation of the ribbon ρ_i will be used. However, when referring to a specific thread the complete notation of ρ_i^j will be used.

A scheduling algorithm A schedules threads, selecting which thread is allowed to execute on the processor at any time. For a schedule to be valid, all threads must complete before the deadline of the job they belong to. A scheduler may preempt threads in one of two ways: *thread level* or *job level*. A thread level preemption is between threads of the same task, such as ρ_i of τ_i being suspended when ρ_j is released by τ_i and selected for execution. A job level preemption occurs when a thread of one task ρ_i of τ_i replaces an executing thread of another task ρ_j of τ_j.

Tasks	Task	Ribbons
$\tau_i \in \tau$	$\tau_i = (p_i, d_i, \rho_i)$	$\rho_j \in \rho$

Cache Size (Number of Lines)	Instruction Time	BRT
l	$\mathbb{I}$	$\mathbb{B}$

TABLE I: Summary of Model Parameters

A. Assumptions and Processing Model

In the proposed model, a task contains multiple ribbons, execution of a job begins with an *initial thread* of the *initial ribbon*. Initial (and subsequent) threads may release threads from any ribbon of their task. For the sake of limited analysis, tasks and their thread releases are restricted. A job release of a task τ_i spawns exactly m threads of an initial ribbon ρ_i. Subsequent thread releases are not permitted. Heterogeneous ribbons and variable thread releases requires additional research.

Jobs are executed on a single processor with an instruction cache of l lines, where each line can store exactly one

instruction. Throughout the paper we assume the cache is direct-mapped, assigning exactly one cache line to a memory address. Each instruction completes in the same $\mathbb{I}$ 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 $(\mathbb{I}+\mathbb{B})$.

A single cycle per instruction (CPI) and direct-mapped cache is used to simplify the presentation of the analytical methods and evaluations. These simplifications are also found in other works. A constant CPI is used in PROARTIS [10] for illustration, when practically applied a lookup table for CPI values would be used. Such a lookup table applies to the analysis proposed in Section V. Similarly, the block reload time $\mathbb{B}$ may be shared amongst instructions loaded simultaneously on architectures where cache blocks contain multiple instructions. Direct-mapped caches simplify the overview of CRPD techniques in Altmeyer's [11] survey. However, each method can be augmented to utilize associative caches when the replacement policy is restricted. In this work, associative caches and their replacement policies require careful consideration and are reserved for a later extension.

B. Inter-Thread Cache Benefit & Prior Research

When a job is scheduled to execute on the processor, the object of the job's task is copied into main memory. Additional memory may be reserved or requested by each job. The range of valid memory locations a job may utilize is referred to as the memory *address space* of the job.

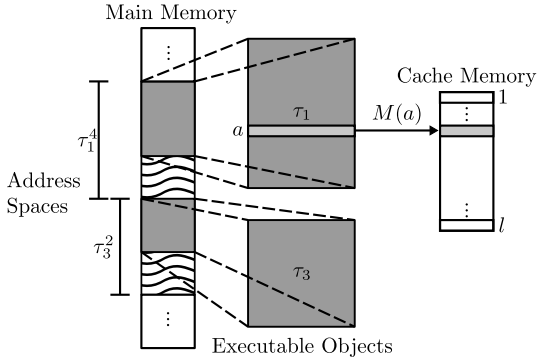


Fig. 1: Address Spaces of Jobs for τ_1 and τ_3

In Figure 1, the address spaces of the fourth job of task one and the second job of task three are shown in main memory. The shaded area is the copy of the executable object, and the sinuous area is the additional reservation made by the job.

Threads share the address space of their job. A thread ρ_j that resides in the memory space of a job of task τ_i *belongs* to the task; written $\rho_j \in \tau_i$. Instructions of a ribbon/thread are referred to by their absolute address a. The cache block that an address a maps to is given by the function $M(a)$.

When a thread ρ_k executes along a fixed path without interruption by preemption, an instruction access that results in a cache miss is called an *opportunity instruction*, or simply an *opportunity*. Similarly, during uninterrupted execution, any instruction access that hits the cache is called an *expected instruction* or an *expectation*.

When multiple threads are executed, the execution time of one or more threads may be influenced by cache interactions. When a thread ρ_j preempts a thread ρ_k, ρ_j may evict cache lines of ρ_k placed there. If those evicted cache lines correspond to expected instructions, ρ_j will increased ρ_k's execution time since ρ_k must now pay $\mathbb{B}$ for each evicted line. Conversely, a thread ρ_j may unexpectedly place opportunity instructions of ρ_k in the cache during a preemption of ρ_k, reducing ρ_k's execution time.

Inter-Thread Cache Benefit: Thus, the inter-thread cache benefit for a thread of ρ_j is the speed-up of $\rho_j \in \tau_i$ due to the conversion of opportunities into expectations by the placement of values in the cache from a thread of $\rho_k \in \tau_i$ when ρ_k is scheduled before ρ_j.

Tasks	Task	Ribbons	Thread Releases
$\tau = \{\tau_1\}$	$\tau_1 = (p_1, d_1, \rho_1)$	$\rho = \{\rho_1\}$	$m = 2$

Cache Size (Number of Lines)	Instruction Time	BRT
$l = 200$	$\mathbb{I} = 1$	$\mathbb{B} = 10$

TABLE II: Example Model Parameters

Using the model parameters in Table II, an example ribbon ρ_1 releasing two threads is presented as a control flow graph [12] (CFG) in Figure 2. The purpose of this example is to clarify the inter-thread cache benefit and expose the pessimism in the existing WCET, CRPD, and scheduling analysis techniques.

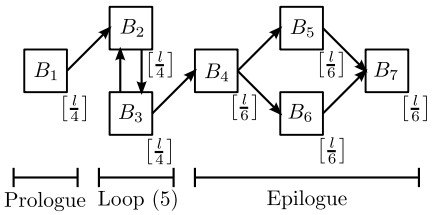


Fig. 2: Control Flow Graph for: ρ_1

The CFG connects serialized sets of instructions, called basic blocks, by their logical control flow through the ribbon. In the figure, below each basic block (in square brackets) is the length of the block as a fraction of the cache size l. The parenthesized value at the bottom of the figure indicates the maximum number of iterations the loop will execute.

The ribbon ρ_1 is analyzed by the WCET calculation methods of Arnold [13] and Mueller [14]. CRPD costs are determined using Lee et al.'s [15] useful cache block (UCB) technique. Although simpler and less accurate than modern techniques, these methods were chosen for illustrative purposes and their continued use in subsequent works.

A necessary step in WCET calculation is the categorization of instructions, such as *must-miss* and *first-miss*. A must-miss never hits the cache. A first-miss always hits the cache after its initial miss. To find first-miss instructions the CFG is searched iteratively looking for return paths. Only instructions with return paths are candidates for first-misses. Table VII (found in the appendix) presents the cache mapping and categorizations.

Lee et al.'s [15] useful cache block (UCB) approach to CRPD calculation borrows the iterative return path approach. From Figure 2, the only candidates for first-miss and UCB instructions are contained in basic blocks B_2 and B_3. No other blocks have a return path and would be categorized as must-

miss, and not useful.

1) WCET: Using these categorizations and the loop bound, the worst case execution time of ρ_1 is the sum of the execution times of the prologue, the entry executions of B_2 and B_3, the repetitions of B_2 and B_3, and the epilogue. Table III gives the intermediate values; the total execution time taking into consideration reloads is: $\frac{l(\mathbb{B}+\mathbb{I})}{4} + \frac{2l(\mathbb{B}+\mathbb{I})}{4} + \frac{8l(\mathbb{I})}{4} + \frac{3l(\mathbb{B}+\mathbb{I})}{6} = \frac{l(5\mathbb{B}+13\mathbb{I})}{4} = 3150$

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

TABLE III: Segment WCET

Under the classical model, two synthetic tasks are created for the two threads of ρ_1. Assigning the WCET of 3150 to both synthetic tasks, the total execution requirement for the one-task system is 6300 every p_1 time units.

However, this is overly pessimistic. The worst possible execution scenario and schedule for the two threads is the sequential execution of ρ_1^1 followed by ρ_1^2, where ρ_1 takes the "high" road executing B_5 and ρ_1^2 takes the low "road" through B_6. This maximizes the number lines ρ_1^2 will place in the cache. Even so, blocks B_2, B_3, B_4 are present in the cache when ρ_1^2 reaches them.

Evaluating the worst case schedule, the WCET of ρ_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 task execution requirement is $3150 + 1400 = 4550$ cycles, less than the 6300 cycles calculated from the synthetic task analysis and application of the Arnold and Mueller approaches. Figure 3 illustrates the worst possible schedule including a summary of cache contents at $t = 3150$, compared to the WCET bound calculated from synthetic tasks.

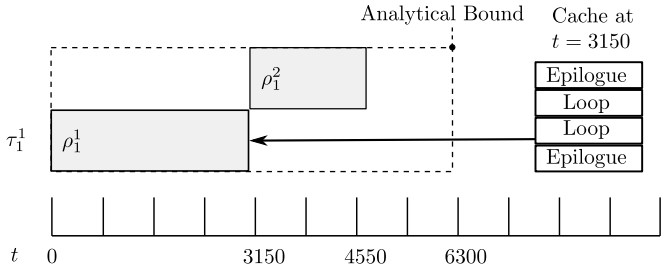


Fig. 3: Worst Schedule of τ_1

2) CRPD: CRPD is an analytical technique that 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 resumption the preempted task must pay the BRT for each invalidated cache block.

One method for CRPD calculation is the Lee et. al [15] useful cache block (UCB) approach. A UCB is "a cache block that contains a memory block that may be referenced before being replaced by another memory block." CRPD for a task is limited by the number of UCBs within it.

From Figure 2 there are two basic blocks that contribute UCBs to the thread ρ_1: B_2 and B_3. Applying Lee's method, the CRPD of a preemption of ρ_1 is $\frac{2l(\mathbb{B}+\mathbb{I})}{4} = 1100$. However, this bound is overly pessimistic.

By construction (and shown in Table VII) once the "Loop" instructions are cached they cannot be invalidated. If ρ_1^1 were to be preempted after the first iteration of the loop, the instructions of the loop body (B_2 and B_3) would be cached in parts 4-10. No other instructions of ρ_1^2 map to those cache lines, and cannot invalidate them. Furthermore, there is no schedule of ρ_1^1 and ρ_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 [16], UCB-Union, and UCB-Union Multiset [11] 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 [13] and Mueller [14] approaches play a role in subsequent WCET methods and none incorporate the inter-thread cache benefit.

III. BUNDLE SCHEDULING ALGORITHM

To allow the inter-thread cache benefit to be realized the BUNDLE scheduling algorithm selects threads to execute (not jobs). A ready thread belongs to a group named a *bundle*. A bundle is always associated with a *conflict free region*: a selection of nodes and edges from a ribbon's CFG with a single entry instruction where no two instructions map to the same cache line, guaranteeing no evictions occur between threads of the same bundle. A formal description and method for extracting conflict free regions is described in Section V.

Scheduling of threads is straightforward. There is one active bundle at any time, only threads of the active bundle are allowed to execute. Within the active bundle, thread execution order is arbitrary and preemptions are permitted.

A thread enters a bundle by attempting to execute the entry instruction of the bundle's conflict free region. A thread leaves a bundle by attempting to execute the entry instruction of a different conflict free region. Leaving (or entering) a bundle causes a thread to block. Only when the active bundle is empty does it become inactive, at which time a new bundle is selected and becomes active.

Pseudocode is given for BUNDLE in Figure 4. The preconditions for scheduling are 1.) the set of ready threads R is populated 2.) all threads $\rho^i \in R$ are awaiting to execute the same instruction 3.) the set of entry instructions for all conflict free regions Y have been precomputed.

Conflict free regions are identified by an entry instruction, it is natural to use the address of that instruction y as the index for the bundle. When a thread ρ^k leaves the active bundle A, it is placed in the blocked bundle set B indexed by the address of the entry instruction y, ie. $B[y] = \{\rho^j, \rho^k, ...\}$. A new active bundle is selected when the all threads have been blocked (the number of blocked threads = $|R|$).

Line 8 of Figure 4 presents the unique problem of anticipating execution. No existing hardware platform we are aware of provides a method for determining which instruction will execute next. To be immediately applicable, ribbons

```
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
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:        end if
16:    end while
17: end procedure
```

Fig. 4: BUNDLE Scheduling Algorithm

must be modified to include synchronization calls (similar to taking a semaphore) before every entry instruction $y \in Y$. This modification communicates the thread's intent to enter a new bundle. Manipulating a ribbon's source is not always practical, a hardware mechanism assisting the scheduler would be preferable. However, the discussion and description of such a potential hardware mechanism is beyond the scope of this paper and left for future work.

IV. Conflict Free Regions

BUNDLE requires the set of entry instructions for all conflict free regions to schedule threads. Timing analysis in Section V demands those conflict free regions have calculable execution time bounds for m threads. This section details the methods for extracting conflict free regions. Before providing the methods for extraction, the definitions and concepts upon which they rely are introduced.

Control Flow Graphs of Ribbons: CFGs [12] are defined as "... a directed graph in which the nodes represent basic blocks and the edges represent control flow paths", where a basic block "is a linear sequence of program instructions having one entry point [...] and one exit point". To simplify the presentation of conflict free regions, single instructions are used as basic blocks. For brevity, the notation G_ρ will be used for the control flow graph of a ribbon ρ.

A control flow graph $G_\rho = (V, E)$ is a weakly connected directed graph. The set of vertexes V are instructions, containing the entry instruction s of ρ and all reachable instructions from s. The set of edges $E = \{(u, v) | u, v \in V\}$ are the changes of control, where $(u, v) \in E$ if u can immediately precede v during the execution of a thread of ρ.

Regions of a Control Flow Graph: a region is a selection of the vertexes and edges of a CFG G. When extracting a region from G, the graph's connectivity is preserved. I.e, two vertexes connected in G must also be connected in any region containing both.

More formally, for a region $U = (V', E')$ of a control flow graph $G = (V, E)$, where $V' \subset V$ and $E' \subset E$. For all pairs of vertexes $(u, v) \in V'$, $(u, v) \in E \iff (u, v) \in E'$. Regions contain an entry instruction $s \in V$ that is weakly connected to all other vertexes in V'.

Conflict Free Region: a region of $U = (V, E)$ of G_ρ is conflict free if no two instructions of U utilize the same cache block.

$$\forall u, v \in V, u \neq v \Leftrightarrow M(u) \neq M(v)$$

An **intra-thread cache conflict** is an eviction that may occur during the non-preempted execution of a thread. For $G_\rho = (V, E)$, instructions $a, b \in V$, $a \neq b$ and valid path $\pi = \langle a, ..., b \rangle$, b is an intra-thread cache conflict if $M(a) = M(b)$.

For all paths starting with an instruction s, there may be multiple intra-thread cache conflicts. We define the *next* intra-thread cache conflicts to be the conflicts "closest" to s.

Next Intra-Thread Cache Conflict: for $G_\rho = (V, E)$, and instruction $s \in V$, a next intra-thread cache conflict is an intra-thread cache conflict u reachable by a path $\pi = \langle s, ..., t, u \rangle$ containing no cache conflicts between any two vertexes $(a, b) \in \pi'$ for $\pi' = \langle s, ..., t \rangle$.

Next Intra-Thread Cache Conflicts: is the set of all possible u values that are a next intra-thread cache conflict from s. The set is given by $p(s)$.

Figure 5 illustrates the next intra-thread conflicts for a portion of the CFG of a ribbon from instruction s. Each vertex contains the index of the cache line the instruction maps to, some are named (e.g., w, t). Executing a thread starting with s, u could evict t or v could evict s. Although t could evict w (since w must execute before s), t is not a next conflict starting from s. There are only two conflicts from s, $p(s) = \{u, v\}$.

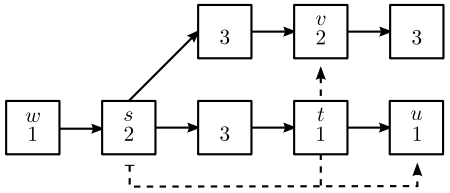


Fig. 5: Next Intra-Thread Conflicts From s

An algorithm to calculate $p(s)$ is included in Figure 6. It relies on a separate procedure named *paths of conflict* (*poc*), that returns a set of paths each starting with instruction s terminating with first conflict on that path. Exploring all possible paths recursively, *poc* terminates a path search when a conflict is found – pseudocode is presented in Appendix C. It is the set of conflicts, not paths, that $p(s)$ returns: the last element of each path found by $poc(s)$.

```
1: procedure p(s)
2:     R ← ∅
3:     ℙ ← poc(s)
4:     for all P ∈ ℙ do
5:         R ← {R ∪ P.last}
6:     end for
7:     return R                  ▷ set of next intra-thread conflicts
8: end procedure
```

Fig. 6: Definition of Next Intra-Thread Cache Conflicts $p(s)$

An **inter-thread cache conflict** is a possible eviction due to the execution of multiple threads of the same ribbon. For

$G_\rho = (V, E)$, instructions $a, b \in V$, $a \neq b$, a and b are inter-thread conflicts of each other if $M(a) = M(b)$.

Next Inter-Thread Cache Conflict: for $G_\rho = (V, E)$, and instruction $s \in V$, a next inter-thread cache conflict from s is an instruction t where $M(t) = M(u)$ with valid paths $\pi_i = \langle s, ..., t \rangle$, $\pi_j = \langle s, ..., u \rangle$ and no other conflicts exist between elements of either path: $\forall (x, y \in (\pi_i \cup \pi_j) - \{t, u\})\ M(x) \neq M(y)$.

Next Inter-Thread Cache Conflicts: is the set of all possible t values that are a next inter-thread cache conflict from s. The set is given by $P(s)$.

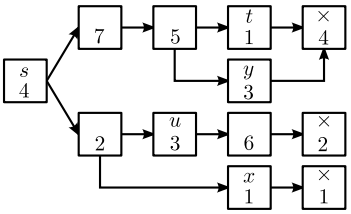


Fig. 7: Next Inter-Thread Conflicts From s, $P(s) = \{t, u, x, y\}$

By definition, an intra-thread cache conflict is an inter-thread conflict, since the conflicting instruction $u \in p(s)$ conflicts with some instruction t on the path π from s to u : $\pi = \langle s, ..., t, ...u \rangle$. For two threads ρ^1 and ρ^2 executing from s, ρ^1 may cache t and ρ^2 evict t by executing u. Similarly, ρ^2 may cache u which may be evicted by ρ^1 executing t, making t and u inter-thread cache conflicts.

Figure 7 illustrates the relationship between intra and inter-thread cache conflicts, a relationship that is leveraged to extract conflict free regions. The set of intra-thread cache conflicts are marked with an $\times$. The set of inter-thread cache conflicts are marked $\{t, u, x, y\}$. Figure 8 depicts the largest region of G_ρ with no inter-thread cache conflicts that preserves the connectedness of G_ρ.

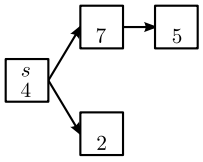


Fig. 8: Region of G_ρ Limited by Inter-Thread Cache Conflicts

An algorithm to find the set of next inter-thread cache conflicts $P(s)$ is given in Figure 9. It utilizes the paths of conflict procedure $poc(s)$ to establish a search limit for conflicts between all paths, since $poc(s)$ returns the next intra-thread cache conflicts which must also be inter conflicts. When an inter-thread conflict is found across two paths P and Q between instructions u and v, the paths are cut (shortened) at those instructions. Conflicts found by this method are not guaranteed to bound the largest region.

Given the definition of conflict free regions, it is clear their boundaries are established by the set of next inter-thread cache conflicts. Extracting conflict free regions is an iterative process, starting with the set of next inter conflicts $P(\rho_i)$ from the ribbon's entry point ρ_i. The set of next inter conflicts identify the entry instructions for the next conflict free region. To extract a conflict free region U, vertexes and edges are added to U by a depth first search from ρ_i to all vertexes $y \in P(\rho_i)$.

```
1:  procedure P(s)
2:      ℙ ← poc(s)
3:      for all (P, Q) ∈ ℙ do
4:          for all u ∈ P, v ∈ Q do
5:              if M(u) = M(v) then
6:                  P ← subpath(P, s, u)
7:                  Q ← subpath(Q, s, v)
8:              end if
9:          end for
10:     end for
11:     return {P.last | P ∈ ℙ}
12: end procedure
```

Fig. 9: Definition of Next Inter-Thread Cache Conflicts $P(s)$

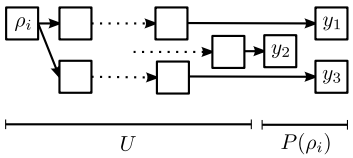


Fig. 10: Extraction of the First Region U from G_ρ

In Figure 10, the next inter-thread cache conflicts from ρ_i are labeled $P(\rho_i) = \{y_1, y_2, y_3\}$. The conflict free region $U = (V, E)$ contains none of those conflicts: $V \cap P(\rho_i) = \varnothing$. However, each y value identifies an entry instruction for a subsequent region. Extraction is repeated for each new region until the terminal instruction is reached.

V. Timing Analysis

WCET+O analysis for a task with one ribbon ρ releasing m threads per job depends on the structure of the program, conflict free regions and BUNDLE's scheduling decisions. The result is a time bound $c(m)$ for all m threads to complete their execution.

In Figure 11, the ribbon ρ, that starts with s and ends with t, has been divided into conflict free regions. Attached to each region are two values, the single-thread cost $m = 1$ and the additional-thread cost: the difference in execution time between $m = 2$ and $m = 1$ threads. Each conflict free region is treated as a single node in a new graph that preserves the edges between regions from the original program, for clarity this new graph is referred to as the *region graph*.

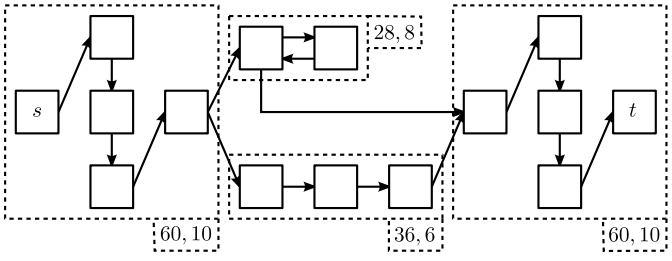


Fig. 11: Conflict Free Regions and Bounds for ρ

Vertexes of the region graph are identified by their initial instruction, either s the entry point of the ribbon or $y \in Y$ from the static analysis. A path π through the region graph always begins with s and terminates with the region containing the terminal instruction t: $\pi = \langle s, y_1, y_2, ... \rangle$.

For each region y, the WCET+O of m threads (while all other threads are blocked) is given by the function $c_y(m)$ (to be defined more formally later). The WCET+O of a path is then $c_\pi(m) = \sum_{y \in \pi} c_y(m)$. To calculate the WCET+O of m threads over the region graph a worst-case selection of paths and number of threads $\langle \pi, n \rangle$ is recorded in the multiset Π_S, supported by all distinct paths Π, denoted $\Pi_S \subseteq \Pi$.

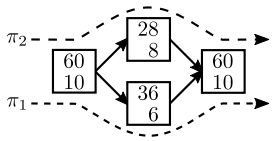
The cardinality of Π_S is strictly m, ie. $m = |\Pi_S|$. A path π_i occurring exactly n times in Π_S is denoted by $\pi_i \in^n \Pi_S$. The n threads traversing π_i will share bundles for each region $y \in \pi_i$. For a path selection Π_S, completion of m threads is bounded by Equation 1 below.

The number of occurrences k, of a path $\pi_i \in^k \Pi_S$ is determined by the contribution to the bound to complete all threads $c(m)$. There must not exist a selection $\Pi_{S'}$ of k values that yields a greater bound $c(m)$. Construction of Π_S is given by Equation 2. Alternatively, paths may be added to Π_S iteratively, finding the largest cost increase of any path $c_\pi \in^i \Pi_S$ to $c_\pi \in^{i+1} \Pi_S$ until $|\Pi_S| = m$.

$$c(m) = \sum_{\pi_i \in \Pi_S} \sum_{y \in \pi_i} c_y(\{n \mid \pi_i \in^n \Pi_S\}) \quad (1)$$

$$\Pi_S = \underset{\Pi_S \subseteq \Pi}{\operatorname{argmax}} \left\{ \sum_{\pi_i \in \Pi_S} \sum_{y \in \pi_i} c_y(\{n \mid \pi_i \in^n \Pi_S\}) \right\} \quad (2)$$

Figure 12 is the region graph of ρ from Figure 11. The dashed lines are the two possible paths through the ribbon and their assignments as the worst case paths. If $m = 4$ then $c(m) = 156 + 148 + 28 + 28 = 360$, π_1 only occurs once in Π_S since its subsequent cost is only 26. The bound $c(m)$ is safe by construction, accounting for the worst possible control flow scenario for each thread.



$c(1) = 156 \quad \Pi_S = \{\pi_1\}$
$c(2) = 304 \quad \Pi_S = \{\pi_1, \pi_2\}$
$c(3) = 332 \quad \Pi_S = \{\pi_1, \pi_2, \pi_2\}$
$c(4) = 360 \quad \Pi_S = \{\pi_1, \pi_2, \pi_2, \pi_2\}$

Fig. 12: Region Graph and Worst Case Paths of ρ

This example exposes BUNDLE's sub-optimal behavior. Since the bundle (conflict free region) execution order is unknown, bundles may be scheduled across the same region multiple times without benefiting from cache reuse.

To formally calculate $c_y(m)$ for a conflict free region U with entry instruction y, the following assumptions are made.

1) The region $U = (V, E)$ has an entry instruction $s \in V$.
2) All m threads are ready and waiting to execute s.
3) Any thread that attempts to execute an instruction $y \in P(s)$ is blocked.
4) Preemptions between threads take no time.
5) Loops have predetermined iteration bounds.

In addition to these assumptions, a time bound calculation relies upon a single logical *structure* within the region. Separated into three types: linear, branching, and looping; their descriptions are given in the following subsection. Each region is divided into smaller regions containing one structure maintaining the connectedness of G_ρ.

Taking an iterative approach, structures are extracted from the entry instruction s of a conflict free region U; the first structure is U_i detected along with a set of boundary instructions K_i, which serve as entry points for subsequent structures. This process continues until all vertexes of U are assigned to a structure such that $U = \bigcup_i U_i$. Since U is a conflict-free region, all structures U_i obtained from U are also conflict-free regions (albeit smaller). The set of entry points for all structures form the set of entry points used by BUNDLE, $Y = \bigcup_{U \in G_\rho} \{\bigcup_{K_i \in U} K_i\}$.

Given the restricted paper length along with established techniques of pathfinding [17] and loop detection [18] we describe only the requirements of structures, not the methods for their extraction.

Linear Structure: a linear structure starting with s is a serial set of instructions with no branches. The out-degree of any vertex in the structure is at most one. It terminates at t, an instruction preceding a branching or looping instruction k. The terminating instruction t is within the struture, while the boundary vertex k is without.

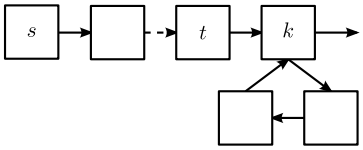


Fig. 13: Linear Structure from s to t Preceding a Loop

Branching Structure: a branching structure contains at least one vertex with outdegree greater than one and no cycles.

A branching structure terminates at a set of instructions T. A vertex $t \in T$ is defined as an instruction that precedes a vertex within a cycle, or has outdegree zero. When a vertex t is determined to be in T, all outgoing edges and paths are pruned from t. Immediate successors of t are added to the set of boundary vertexes $K = \{k \mid (t, k) \in E\}$. Terminal instructions are included within the structure, while boundary instructions are not.

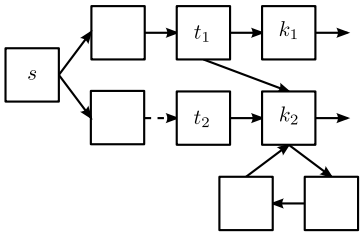


Fig. 14: Branching Sub-Graph from s to $T = \{t_1, t_2\}$ with boundary vertexes $K = \{k_1, k_2\}$

Looping Structure: a looping structure contains a cycle starting with s and all instructions on a path that returns to s. It contains no vertexes outside of the cycle. The inner structure of the loop is restricted, no path from a vertex within the cycle leaves the cycle without passing through s. This restriction is met by precluding GOTO and LONGJMP instructions in tasks. However, within the cycle there may any number of linear, branching, or inner looping structures.

The following theorems and proofs provide a method for calculating a WCET+O value for a linear structure and an arbitrary number of threads $m \in \mathbb{N}^+$. The setting is a single

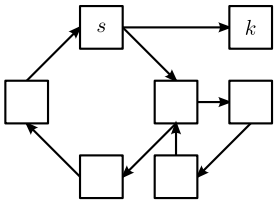


Fig. 15: Looping Structure from s with Boundary k

conflict free region $U = (V, E)$ containing a single structure with entry instruction s, a set of next conflicts $Y = P(s)$. All m threads are scheduled by BUNDLE, blocked waiting to execute s, and will block when attempting to execute any $y \in Y$.

Theorem 1: For m threads that are blocked waiting to execute the entry instruction s of a conflict free region $U = (V, E)$, an instruction $v \in V$ cannot be evicted during the execution of any thread across the region, if any thread is blocked before executing an instruction $y \in Y$.

Proof: By definition of a conflict free region, $\forall u, v \in V \ M(u) \neq M(v)$ if $u \neq v$. Consider a cached instruction v, if the execution of $u \in V$ evicts v then $u \neq v$. Further, $M(u) = M(v)$ contradicting the definition of a conflict free region. Therefore, v cannot be evicted by execution of any $u \in V$. ■

Corollary 1.1: During the execution of a conflict free region by BUNDLE, any instruction $v \in V$ can be loaded into the cached no more than once for any number of threads.

Time Bound for Linear Structures: A conflict free region $U = (V, E)$ containing a single linear structure has a starting instruction s and terminal instruction t. With no branches, there is a single path $\pi = \langle s, ..., t \rangle$ in U. The length of this path is referred to as $L = |\pi|$.

Theorem 2: For a conflict free region U with linear structure and m threads waiting to execute the starting instruction s, an upper bound on the execution time from s to t for all threads is: $c_s(m) = L(\mathbb{I} \cdot m + \mathbb{B})$

Proof of Theorem 2: Each of the m thread executes L instructions since there are no alternative paths from s to t. By Corollary 1.1, at most one of the m threads will cache each of the L instructions taking $L \cdot \mathbb{B}$ time. Execution of L instructions by m threads takes $L \cdot \mathbb{I} \cdot m$ time. Combining the time required to cache and execute, yields the bound of $c_s(m) = L(\mathbb{I} \cdot m + \mathbb{B})$. ■

Branching and Looping Structures: Timing bounds and proofs for branching and looping structures follow the same general structure. Calculating the worst case execution scenario for execution and cache loads separately. The worst case execution scenario occurs when all threads take the longest path. While the worst scenario for cache misses occurs when the threads cover all paths, caching every value. Theorems and their proofs for branching and looping structures are found in Appendix E supported by Appendix D.

VI. Evaluation

Two comparative evaluations were performed. The first investigates the static timing analysis bound for a multi-threaded program from the classical and integrated perspectives. The second presents BUNDLE's run-time overhead against two other thread level schedulers.

Evaluation of Static Analysis: The object being compared is a parallel program using the POSIX Thread (pthread) library to estimate the ratio of a circle's circumference and diameter (π). The structure of the program is linear: initializing parameters, releasing threads, and accumulating the results. The full source is listed in Appendix F.

Initialization and accumulation are handled by a `main` initial thread. The contribution of the initial thread to execution time and cache contents remains constant, so it is ignored. Analysis is limited to the m threads of the ribbon ρ.

The ribbon ρ is structured as a prologue, loop body, and epilogue. When compiled, the prologue and epilogue are serialized sets of instructions which correspond to linear structures. The loop body contains branches creating looping and branching structures.

A representative value is calculated for the WCET of ρ for m threads in terms of $\mathbb{I}$ and $\mathbb{B}$. Using Arnold [13] and Mueller's [14] approaches, the prologue and epilogue are considered "never cached" instructions. The loop body is categorized as "first miss" with some instructions that "may" and others that "must" be cached. Where the analysis finds "may be cached" instructions in the loop body we treat them as "must be cached" – reducing the number of cache misses in favor of the classical perspective. Treating each thread as a distinct task, the WCET bound for a single release of ρ is referred to as c_1, the combined WCET is $c_1 \cdot m$ for all threads.

c_1	$86\mathbb{IB} + 35\mathbb{I} + (i-1)(100\mathbb{I} + 8\mathbb{B})$
γ_1	$65\mathbb{B}$
$c(m)$	$13\mathbb{I}m + 105\mathbb{B} + 100im\mathbb{I} + 8i\mathbb{B}$

TABLE IV: Bounds in Terms of Maximum Iterations i

A similar representative value for the CRPD is determined by considering the useful cache blocks [15] of the ribbon. Clearly none of the instructions in the prologue or epilogue are useful. However, all of the instructions within the loop body are useful, they are also evicting. The bound on the CRPD is the product of the number useful cache blocks and the block reload time. Preemption costs follow Luniss [19], incrementing the WCET of each job by the CRPD bound.

The number of cache blocks l is selected such that the loop body of ρ cannot be contained in a single conflict free region, ensuring all structure types are present in the integrated analysis, and benefiting the classical perspective. Performing the classical analysis produces the single synthetic task WCET c_1. From the integrated analysis, $c(m)$ is the WCET+O of m threads scheduled by BUNDLE. Parameterized by the CPI ($\mathbb{I} = 1$), BRT ($\mathbb{B} = 10$) and loop iterations (i), Table IV and Figure 16 summarize the results.

Table V compares the effect of preemptions on the WCET and cache overhead for the classical approch to the WCET+O for BUNDLE. For the integrated approach, preemptions are restricted by BUNDLE and accounted for in the WCET+O; which is why the combined execution time for the integrated approach remains constant. The number of threads is fixed at $m = 2$, increasing m only exaggerates the difference between approaches.

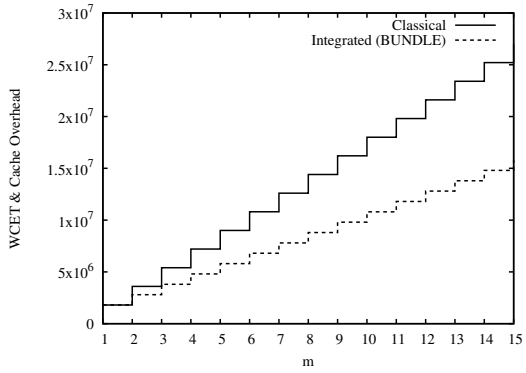


Fig. 16: Comparison of WCET for m, c_1 Tasks and $c(m)$ for m Threads when $i = 1000$

Preemptions		1	5	10	15
Method	Classical	181365	183965	187215	190465
	Integrated	101143			

TABLE V: WCET + Preemption Cost When $i = 10000$

The comparison of WCET bounds and preemption costs illustrates one benefit of the integrated perspective. Although the setting is restricted to a single ribbon with multiple threads, the implications for broader application are clear. Combining timing analysis with an understanding of the scheduling algorithm and cache behavior produces a substantially lower WCET and cache overhead bound.

Evaluation of Runtime Overhead: Complementing the evaluation of static analysis is a runtime comparison of `BUNDLE` against two other thread scheduling algorithms. The first, named `seq`, assigns each thread the same static priority. One thread is selected to execute and runs until completion, at which point another thread is selected to run, the process is repeated until the supply of threads is exhausted. The `random` algorithm executes a random number of instructions of a thread before preempting it, choosing an arbitrary thread to run in its place, execution continues until all threads have completed.

The purpose of this evaluation is to determine the potential runtime benefit of scheduling tasks based upon conflict-free regions (i.e., `BUNDLE`) versus a non-conflict-aware approach. To obtain an evaluation, we implemented a path trace simulator (available at github.com/ctessler/pathsim.git). It takes generated programs as input, tracing the execution of multiple threads using one of the three scheduling algorithms. When a thread reaches a branching instruction, execution continues on a randomly selected path. Similarly, when a looping instruction is reached the number of loops is randomly selected from the provided loop bound. Executed instructions are counted (not timed) and the simulator does not incorporate a pipeline.

Mälardalen Real-Time Research Center's WCET benchmark suite was used to generate the input to the path simulator. An average number of branching statements, looping directives, program length, and basic block sizes were derived from the suite. These values became input to a generator of 100 programs, which varied the values using a Gaussian distribution. Each program was executed a for a given cache size $l = (64, 128, 256, 512)$, number of threads $m = (1, 2, 4, 8, 16)$ and scheduler: (`BUNDLE`, `seq`, `random`). Average instruction counts, cache misses, and context switches were collected from the 7500 runs and summarized in Figure 17 and Table VI.

In terms of runtime cache misses, `BUNDLE` dominates `seq` and `random` in all circumstances. It strictly dominates when the number of threads is greater than one, and the number of cache lines is less than the program length. Scheduling a single thread requires no decisions – each of the algorithms will exhibit the same number of misses; so, the single threaded results are ignored. Similarly, the number of misses will always be zero when the cache size is greater than the program length (as is the case when the cache size is 1024 lines); so, those results are also ignored.

`BUNDLE` outperformed the other algorithms most significantly, when the cache size was nearly two thirds the average program length; Figure 17a presents the comparison. As the number of threads are increased the number of misses remains constant, effectively lowering the cache miss rate.

An inverse relationship holds as the cache size is increased in Figure 17b. For a specific number of threads, increasing the number of cache lines benefits each of the algorithms. While `BUNDLE` maintains its dominance in terms of cache misses and cache miss rate, the relative benefit naturally decreases as the number of cache lines increases.

The lower miss rate comes at a cost: an increased number of thread-level context switches, shown in Figure 17c. However, there is a significant difference between thread-level and task-level context switches. Thread-level switches are designed to be less costly.

Table VI conveys the tension between the cache size, number of threads in the task, and the execution time of a thread-level context switch. Each cell value contains the minimum number of instructions for each context switch before `seq` will dominate `BUNDLE` in terms of the overhead. The cell values are computed by taking the difference in cache misses from the two scheduling algorithms and dividing the difference by the block reload time $\mathbb{B} = 10$.

Cache Lines	64	128	256	512	1024
Threads					
1	0	0	0	0	0
2	248	413	697	1372	0
4	372	618	1046	2057	0
8	434	719	1210	2284	0
16	464	770	1296	2447	0

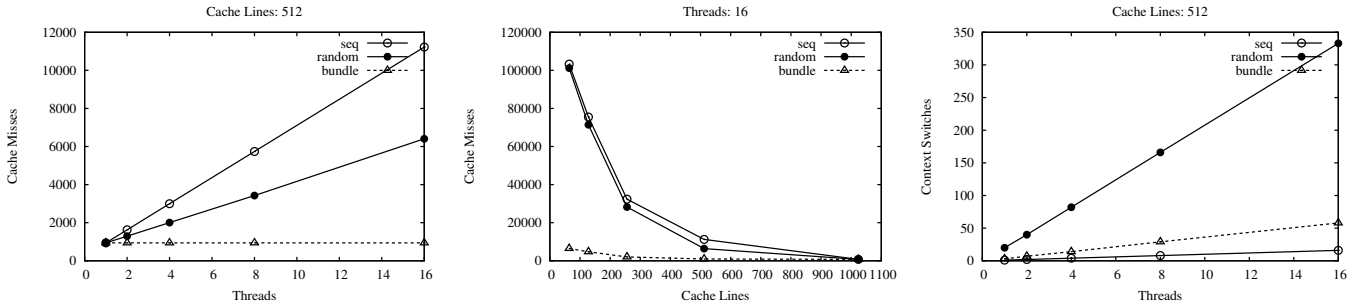
TABLE VI: Minimum Context-Switch Cost (in Cycles) for `seq` to Dominate `BUNDLE`

Across the 100 programs, threads executed an average of 7900 instructions. For the worst-case of two threads and 64 cache lines, a context switch must be 248 instructions before `seq` outperform `BUNDLE`; i.e., each single context switch must be equivalent to 3% of the thread's execution. For the best case, each context switch would have to be equivalent to around 30% of the thread execution! In either case, we believe that such high context switch overheads are not realistic; therefore, the cache reduction of `BUNDLE` is clearly worth the increased thread-level preemptions.

VII. Conclusion and Future Work

Initial experiments on individual tasks demonstrates `BUNDLE`'s immediate benefit. Our future efforts seek to bring `BUNDLE` to an operating system with a shared address space task model (such as μC [20]). Multiple tasks and a complete

Fig. 17: Run Time Overhead Results



(a) Cache Misses vs. Thread Count (b) Cache Misses vs. Cache Size (c) Thread vs. Context Switches

operating system will allow for broader simulation environments (e.g., GRSIM [21] or gem5 [22]) and experimentation.

For multi-threaded task systems the negative view necessitated by the classical perspective results in pessimistic bounds. The classical model lacks the ability to account for the inter-thread cache benefit due to the separate treatment of threads of execution. An integrated approach is necessary to provide safe bounds while taking advantage of instruction caches.

Unfortunately, the completeness of this work is constrained by its length. This affects the inclusion of more modern techniques for WCET and CRPD analysis from the classical perspective for comparison. It also limits the depth and elegance of solutions that can be presented, which results in pessimism and a high degree of computational complexity in several areas.

Thankfully, these omissions provide opportunities for future work. Motivated by the benefit of BUNDLE shown in the evaluations, future work will seek to improve the bundle-selection scheduling rules, increase the size of conflict-free regions (reducing thread-level context switches), and reduce the computational complexity of computing conflict-free regions.

Towards the greater goal of promoting the integrated perspective, the most pressing extension is the generalization of task sets to include multiple tasks and multiple ribbons per task. We also see promise in bringing BUNDLE's principles to multi-core analysis.

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] J. Sun, N. Guan, Y. Wang, Q. Deng, P. Zeng, and W. Yi, "Feasibility of fork-join real-time task graph models: Hardness and algorithms," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 1, pp. 14:1–14:28, Feb. 2016.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] R. Mittermayr and J. Blieberger, "Timing Analysis of Concurrent Programs," in *12th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), vol. 23. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 59–68.

[8] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, Dec 2009, pp. 57–67.

[9] J. M. Calandrino, "On the design and implementation of a cache-aware soft real-time scheduler for multicore platforms," Ph.D. dissertation, Chapel Hill, NC, USA, 2009.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] S. M. Kumari and N. Geethanjali, "A survey on shortest path routing algorithms for public transport travel," *Global Journal of Computer Science and Technology*, vol. 9, no. 5, pp. 73–75, 2010.

[18] P. Mateti and N. Deo, "On algorithms for enumerating all circuits of a graph," *SIAM Journal on Computing*, vol. 5, no. 1, pp. 90–99, 1976.

[19] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis, "Integrating cache related pre-emption delay analysis into edf scheduling," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, April 2013, pp. 75–84.

[20] J. J. Labrosse, *Microc/OS-II*, 2nd ed. R & D Books, 1998.

[21] C. G. AB, *GRSIM Users Manual*. Cobham Gaisler AB, 2015.

[22] 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.

APPENDIX A
ACKNOWLEDGEMENTS

This research has been supported in part by the US National Science Foundation (CNS Grant Nos. 0953585, 1205338, & 1618185) and a grant from Wayne State University's Office of Vice President of Research.

APPENDIX B
EXAMPLE ρ_1 INSTRUCTION MAPPING AND CATEGORIES

The cache assignment guarantees all instructions of a basic block will share their categorization as must-miss, or first-miss. For simplicity the blocks (instead of instructions) are given categories.

Block	Category
B_1	must-miss
B_2 B_3	first-miss
B_4 B_5 B_6 B_7	must-miss

Cache Part n of 12	Blocks		
1	B_1	B_5	B_6
2		B_7	
3			
4	B_2		
⋮	B_3		
11	B_4		
12		B_5	B_6

TABLE VII: Categories from [13], [14] and Cache Assignment

APPENDIX C
PATHS OF CONFLICT

The paths of conflict algorithm in Figure 18 is recursive, exploring all paths from s. Each path of the returned set $\mathbb{P}$ begins with the starting instruction s in $poc(s)$, the paths terminate at a next intra-conflict from s.

It uses a global cache state C, which is treated as an array, indexed by cache block a where $a = M(u)$ and u is an absolute memory address. It stores at a the address of the instruction u. Before each recursive call at Line 17, the cache state is copied for use by the next recursive call, and restored from the copy afterward.

For every iteration of the while loop on Line 7, at least one path is explored until a conflict is reached. Recursive calls are made within the loop, searching at least one more path to a next conflict. Maintaining focus on the while loop, there are two notable cache states with respect to u and a. At Line 11, u has been cached indicating a cycle in the graph has been traversed. At Line 12, u conflicts with a cached instruction.

It is possible that paths return by a recursive call have undetected conflicts based on the current recursive depth. The *double for* loop at Line 24 removes paths that would violate the definition of a next conflict by containing a conflict before its final vertex.

APPENDIX D
BOUNDED CONFLICTS

Loops with inter-thread cache conflicts rely on a different type of inter-thread cache conflict. The definitions of *bounded conflicts* are necessarily provided before the theorems and proofs.

A **bounded intra-thread cache conflict** from a given instruction s up to and including instruction v is an intra-thread cache conflict on any path $\pi = \langle s, ..., v \rangle$.

1: $G = (V, E)$ ▷ From ribbon r
2: C ▷ Initially empty
3: **procedure** $poc(s)$
4: $W \leftarrow \{s\}$
5: $P \leftarrow \varnothing$ ▷ One path
6: $\mathbb{P} \leftarrow \varnothing$ ▷ All paths, return value
7: **while** $|W| > 0$ **do**
8: $u \leftarrow w, w \in W$; $W \leftarrow \{W - w\}$
9: $a \leftarrow M(u)$
10: $P \leftarrow \{P, u\}$
11: **goto** 7 **if** $C[a] = u$ ▷ Loop detected, go to next u
12: **goto** 23 **if** $C[a] \neq \varnothing$ ▷ Conflict, end search
13: $S \leftarrow \{v \mid (u, v) \in E\}$
14: **if** $|S| > 1$ **then**
15: **for all** $v \in S$ **do**
16: $C' \leftarrow C$
17: $\mathbb{P} \leftarrow \{\mathbb{P} \cup poc(v)\}$
18: $C \leftarrow C'$
19: **end for**
20: **end if**
21: **end while**
22: $\mathbb{P} \leftarrow \{\mathbb{P}, P\}$
23: ▷ Remove all paths with earlier conflicts
24: **for all** $P \in \mathbb{P}$ **do**
25: **for all** $T \in \mathbb{P}$ **do**
26: $\mathbb{P} \leftarrow \{\mathbb{P} - P\}$ **if** $T.last \in P$
27: **end for**
28: **end for**
29: **return** $\mathbb{P}$
30: **end procedure**

Fig. 18: Definition of Paths of Conflict $poc(s)$

Bounded Intra-Thread Cache Conflicts: is the set of all possible u values that are bounded intra-thread cache conflicts from s to v. The set is given by $b(s, v)$.

No algorithm for $b(s, v)$ is given, it is seen as a straightforward modification to $p(s)$ and $poc(s)$ that terminates when reaching v instead of a conflict. This set may include more conflicts than the next intra-thread conflicts $p(s)$.

A **bounded inter-thread cache conflict** from a given instruction s up to and including v is an inter-thread cache conflict on a path from $\pi = \langle s, ..., v \rangle$.

Bounded Inter-Thread Cache Conflicts: Are the set of all possible u values that are a bounded inter-thread cache conflict from s to v. The set is given by $B(s, v)$.

The algorithm for $B(s, v)$ is omitted, since it would require no changes to $P(s)$ assuming $poc(s)$ was modified to accept a bound v, as suggested for bounded intra-thread cache conflicts. This set may include more conflicts than the next inter-thread conflicts $P(s)$.

APPENDIX E
TIMING THEOREMS AND PROOFS FOR ADDITIONAL STRUCTURES

Time Bound For Branching Structures: A conflict free region $U = (V, E)$ with a single branching structure has a starting instruction s and a set of terminal instructions T.

With multiple paths $\pi = \langle s, ..., t \rangle$, where $t \in T$. The length of longest path to any $t \in T$ from s is referred to as L

Theorem 3: For a conflict free region $U = (V, E)$ with a branching structure, and m threads waiting to execute s, an upper bound on the execution time from s to $t \in T$ for all threads is:

$$c_s(m) = L \cdot \mathbb{I} \cdot m + |V| \cdot \mathbb{B}$$

Proof of Theorem 3: From Corollary 1.1 at most one of the m threads will cache any $v \in V$, the worst possible case is that all $|V|$ instructions are cached taking $|V| \cdot \mathbb{B}$ time. For execution, the worst case is for all m threads to execute the longest path of length L taking $L \cdot \mathbb{I} \cdot m$ time. Combining the bounds produces $c_s(m) = L \cdot \mathbb{I} \cdot m + |V| \cdot \mathbb{B}$. ■

Time Bound For Looping Structures: A conflict free region $U = (V, E)$ with a single looping structure has a cycle starting with instruction s, and bound on the number of iterations i. There may be multiple distinct cycles from s to s, among those cycles the one with the longest path is referred to as L.

Theorem 4: For a conflict free region $U = (V, E)$ with a looping structure, and m threads waiting to execute s, an upper bound on the execution time for all threads to complete i iterations of the cycle is given by

$$c_s(m) = i \cdot m \cdot L \cdot \mathbb{I} + |V| \cdot \mathbb{B}$$

Proof of Theorem 4: Consider the execution and caching of instructions separately. Since L is the longest path through the cycle and, one cycle executed by one thread can take no more than $L \cdot \mathbb{I}$ time. For m threads and i iterations the upper bound on execution is $i \cdot m \cdot L \cdot \mathbb{I}$.

Cache misses are limited by Theorem 1, since U is conflict free, no instruction can be evicted during the execution of U. Only the initial load of any instruction into the cache need be considered. The number of initial loads is bounded by the total number of instructions in the region which takes $|V| \cdot \mathbb{B}$ time.

Combining the bounds on execution and caching of instructions result in $c_s(m) = i \cdot m \cdot L \cdot \mathbb{I} + |V| \cdot \mathbb{B}$. ■

A Special Case for Looping Structures: Theorem 4 assumes looping structures are contained within a single conflict free region. Looping and branching structures may be divided at boundaries defined by inter-thread conflicts. However, for looping structures, this is not always the case.

When a cycle in G_ρ contains an inter-thread cache conflict, a separate time bound for the cycle U must be calculated. Cycles have the restricted form of an entry instruction s with two outgoing edges, one that enters the cycle and another exiting through the boundary instruction k.

The set of bounded inter-thread cache conflicts, calculated by $B(s, v)$, differ from the set of next inter-thread cache conflicts by including all conflicts on all paths $\pi = \langle s, ..., v \rangle$. The next conflicts, calculated by $P(s)$, are limited to the first conflict on each path and may not reach t. The bounded conflicts $B(s, s)$ are necessary for calculating the bound of m threads over the looping structure U. In this special case, only the entry s and boundary k instruction are added to the entry points Y used by BUNDLE.

Theorem 5: For a looping structure $U = (V, E)$ that contains inter-thread cache conflicts with m threads waiting to execute s, and maximum length of a simple cycle $L = |\pi|, \pi = \langle s, ..., s \rangle$, an upper bound on the execution time for all threads to complete i iterations of the cycle is given by

$$c_s(m) = \mathbb{B}\big(|V - B(s, s)|\big) + i \cdot m\big(L \cdot \mathbb{I} + \mathbb{B} \cdot |B(s, s)|\big)$$

Proof of Theorem 5: Consider the time to cache all instructions of the loop separately from the time to execute a single iteration. The product of the block reload time and number of instructions $|V| \cdot \mathbb{B}$ bounds the time to populate the cache.

For a single iteration of a loop by a single thread, the execution time is bounded by $L \cdot \mathbb{I}$, for all m threads $m \cdot L \cdot \mathbb{I}$. In one iteration, a single thread will incur at most $|B(s, s)|$ evictions. For all m threads there are no more than $m \cdot |B(s, s)|$ evictions per iteration.

Combining the execution time, block reloads, and iterations produces the time bound $i \cdot m\big(L \cdot \mathbb{I} + \mathbb{B} \cdot B(s, s)\big)$ of executing i iterations of the loop after all instructions are cached. Before incorporating the time to populate the cache, the double counting of $|B(s, s)|$ reloads are subtracted from $|V|$. Summing the time to populate the cache and iterate over the loop for m threads yields the bound listed in Theorem 5. ■

APPENDIX F
ESTIMATING π

Listing 1: `ppi.c` a Multi-Threaded π Estimator Using `PTHREAD`

```
#define M 150
#define L 10000

void *part(void *count) {
        double x, y, d;
        int i;
        long *c = (long *) count;
        *c = 0;

        for (i = 0; i < L; i++) {
                x = rand() / (double) RAND_MAX;
                y = rand() / (double) RAND_MAX;
                d = sqrt(x * x + y * y);
                if (d <= 1) {
                        (*c)++;
                }
        }
        pthread_exit((void *) c);
}

int main (int argc, char *argv[]) {
        for (t = 0; t < M; t++) {
                pthread_create(&threads[t], NULL, part,
                     (void *)&count[t]);
        }
        total = 0;
        for (t = 0; t < M; t++) {
                pthread_join(threads[t],
                     (void *) &found);
                total += *found;
        }
        pi = (double) 4 * total / (M * L);

        printf("M:%i L:%i pi =~ %0.05f\n",
               M, L, pi);
        return 0;
}
```