Minimizing Cache Overhead via Loaded Cache Blocks and Preemption Placement

John Cavicchio, Corey Tessler, and Nathan Fisher
Wayne State University
{ba6444, corey.tessler, fishern}@wayne.edu

***Abstract*— Schedulability analysis for real-time systems has been the subject of prominent research over the past several decades. One of the key foundations of schedulability analysis is an accurate worst case execution time (WCET) measurement for each task. In real-time systems that support preemption, the cache related preemption delay (CRPD) can represent a significant component (up to 44% as documented in research literature) [1]–[3] of variability to overall task WCET. Several methods have been employed to calculate CRPD with significant levels of pessimism that may result in a task set erroneously declared as non-schedulable. Furthermore, they do not take into account that CRPD cost is inherently a function of where preemptions actually occur. Our approach for computing CRPD via *loaded cache blocks* (LCBs) is more accurate in the sense that cache state reflects which cache blocks and the specific program locations where they are reloaded.**

Limited preemption models attempt to minimize preemption overhead (CRPD) by reducing the number of allowed preemptions and/or allowing preemption at program locations where the CRPD effect is minimized. These algorithms rely heavily on accurate CRPD measurements or estimation models in order to identify an optimal set of preemption points. Our approach improves the effectiveness of limited optimal preemption point placement algorithms by calculating the LCBs for each pair of adjacent preemptions to more accurately model task WCET and maximize schedulability as compared to existing preemption point placement approaches. We propose an optimal preemption point placement algorithm using dynamic programming. Lastly, we will demonstrate using a case study improved task set schedulability and optimal preemption point placement via our new LCB characterization.

***Index Terms*—cache-related preemption delay, explicit preemption placement, limited preemption scheduling, worst-case execution time, scheduability analysis**

.

I. Introduction

Real-time systems differ from traditional computing systems in that all tasks must complete execution within a specified deadline, otherwise the utility of the result is severely diminished or utterly useless. Ensuring real-time task systems meet their corresponding deadlines is the subject of schedulability analysis. Paramount to schedulability analysis is accurate characterization of WCET. The term WCET represents the worst case execution time of a task. While the debate continues on whether non-preemptive versus fully preemptive execution is more effective, recent research on limited preemption models has shown to outperform non-preemptive scheduling approaches in terms of schedulability and the fully preemptive scheduling approach in terms of the number of preemptions when preemption cost is considered. Non-preemptive execution has the disadvantage of introducing blocking of high priority tasks by lower priority tasks whereas fully preemptive execution has the disadvantage of significant preemption overhead which has shown to be up to 44% [1]–[3] of a tasks WCET.

One of the noticeable contributors to preemption overhead is due to cache related preemption delay (CRPD). CRPD occurs when a task denoted τ_i is preempted by one or more higher priority tasks denoted τ_k whose execution results in the eviction of cache memory blocks that must be subsequently reloaded when task τ_i resumes execution. Limited preemption approaches have the advantage of reduced blocking with a limited number of allowed preemptions while having the advantage of sections of non-preemptive regions (NPRs) reducing the preemption overhead. One promising approach to implementing a limited preemption approach is selecting preemption points for each task subject to the constraint on maximum non-preemptive region execution time Q_i. A paper by Bertogna et al. [4] proposed and realized a linear time algorithm for selecting optimal preemption points for a sequential basic block structure. Basic blocks are the vertices V of a control flow graph (CFG) connected in a sequence by edges E representing the execution sequence of one or more job instructions. A sequential basic block structure implies that conditional logic and branches are fully contained within basic block boundaries. Existing research utilizes pessimistic CRPD costs that effectively limit the effectiveness of preemption point placement algorithms. The primary contributions outlined in this paper include improved accuracy for computing CRPD cost taking into account where preemptions actually occur, and providing an optimal preemption point placement algorithm implemented via dynamic programming using the more accurate CRPD cost. Furthermore, we demonstrate using a case study improved task set schedulability as compared to state-of-the-art methods.

The rest of this paper is organized as follows. First, current research efforts and related work in the areas of cache related preemption delay and limited preemption scheduling are discussed in Section II. Section III describes the real-time task model terminology used in this paper. The enhanced CRPD computation approach is detailed in Section IV. Section V

briefly outlines the integrated WCET/CRPD computation approach incorporated into fixed priority (FP) and earliest deadline first (EDF) schedulability analysis. The enhanced limited preemption point placement algorithm leverages the improved WCET/CRPD computation in a quadratic time algorithm as discussed in Section VI. A case study using well known benchmark tasks illustrating the benefits of our proposed method are presented and summarized in Section VII. Finally, we will offer relevant conclusions along with proposed future work in Section VIII.

II. Related Work

This paper draws from two areas of real-time theory resulting in two significant contributions. Hence, the descriptions of the related works are divided into two separate subsections, namely, CRPD calculation, and limited preemption scheduling.

A. CRPD Calculation

Analyzing the preempted task, Lee et al. [5]–[7] introduced the concept and algorithm for computing the set of useful cache blocks (UCB) for statically addressed instruction and data for direct mapped and set-associative caches. The UCBs of the preempted task are used to compute an upper bound on the CRPD.

Analyzing the preempting task, Tomiyamay and Dutt [8] computed the set of evicting cache blocks (ECBs) via program path analysis formulating an integer linear programming model for direct mapped instruction caches. In a similar fashion, the ECBs of the preempting task are used to compute the CRPD. Formal definitions of UCBs and ECBs were outlined by Altmeyer and Burguiere [9]. The preempting tasks memory accesses quantified as the set of evicting cache blocks will evict useful cache blocks thereby imposing non-negligible CRPD on the preempted task.

Complementary work by Negi et al. [10] and Tan and Mooney [11] compute the intersection of the ECB and UCB sets to achieve tighter bounds on the CRPD computation for direct-mapped and set-associative instruction caches. Staschulat and Ernst [12] realized an improvement in computational complexity at the expense of CRPD accuracy or tightness via a cache state reduction technique for direct mapped instruction caches that was later extended to address set-associative caches.

Likewise, WCET analysis tools use an over-approximation to estimate cache misses and an under-approximation to estimate cache hits. Cache hits are memory blocks that must reside in cache memory hence the term used to describe this set is must cache [9]. Altmeyer and Burguiere [9] addressed this over-approximation issue by introducing the notion of definitely-cached useful cache block (DC-UCB) [9]. The DC-UCB is useful in schedulability analysis to avoid double counting of cache misses resulting from intra-task cache block eviction.

Ramaprasad and Mueller [13] examine the problem of dynamic addressing supporting CRPD analysis. Their algorithm employs memory access patterns to compute CRPD, instead of UCBs. An important distinction to note is that instruction memory accesses are tightly coupled to the control flow graph in contrast to data memory accesses. This isomorphic property means the set of UCBs corresponding to data memory accesses changes more frequently thereby mandating UCB computation at the instruction level.

B. Limited Preemption Scheduling

The motivation for limited preemption scheduling approaches stems from limitations of fully preemptive and non-preemptive scheduling. Fully preemptive scheduling suffers from schedulability degradation due to increased preemption overhead penalties of which CRPD comprises a significant portion. Non-preemptive scheduling suffers from reduced system utilization due to the blocking imposed on high priority jobs. These factors have motivated research on alternative limited preemption scheduling approaches with the goal of achieving higher task utilization and reduced preemption overhead.

One such approach is known as the deferred preemption model. The idea behind the deferred preemption model is to permit a currently executing job to execute non-preemptively for some period of time after the arrival of a high priority job. Two distinct models of deferred preemption have been proposed by Burns [14] and Baruah [15] known as the fixed preemption point model and the floating preemption point model respectively.

In the floating preemption point model [15], the beginning of non-preemptive regions occur with the arrival of a higher priority job. The currently executing job continues executing non-preemptively for Q_i time units or earlier if the job completes execution. The location of the non-preemptive regions is nondeterministic or essentially floating. Baruah's approach [15] computes the maximum amount of blocking time denoted Q_i for which a task τ_i may execute non-preemptively while still preserving scheduling feasibility.

Another limited preemption scheduling technique known as preemption threshold scheduling was proposed by Wang and Saksena [16]. In preemption threshold scheduling, each task is assigned two priority values, namely, a nominal static priority p_i and a preemption threshold Π_i. A task will be preempted only if the preempting task has a nominal priority p_k greater than the preemption threshold Π_i. In the fixed preemption point model [14], a task can be preempted only at a limited set of pre-defined locations. Basically, tasks contain a series of non-preemptive regions. Preemptions are permitted at non-preemptive region boundaries or fixed preemption points.

Two closely related subsequent works implementing a fixed preemption point model for a fixed priority (FP) scheduler were proposed by Simonson and Patel [17] and by Lee et

al. [7] whose objective was to reduce preemption overhead. In Simonsons and Patels approach [17], tasks are sub-divided into distinct non-overlapping intervals, each constrained by the maximum blocking time Q_i that higher priority tasks may be subjected to while preserving task set schedulability. At a location within each interval containing the minimum number of UCBs, a single preemption point is placed. Lee et al. [7] adopted a different method whereby the locations of preemption points are commensurate with the cardinality of the UCB set less than a pre-determined threshold. While both techniques serve to improve preemption overhead over the fully preemptive approach, their heuristic nature is unable to achieve a globally optimal solution. Complementary work by Bril et al. [18] recently integrated CRPD costs into fixed priority preemption threshold schedulability analysis for task sets with arbitrary deadlines. Optimal priority thresholds are assigned via a CRPD cost minimization algorithm.

In contrast, Bertogna et al. [19] achieved an optimal preemption point placement algorithm with linear time complexity. The analysis assumed a pessimistic fixed constant context switch cost at each preemption point equal to the largest preemption overhead experienced by a task. Later work by Bertogna et al. [4] relaxes the fixed constant context switch assumption using more accurate variable preemption overhead cost information via available timing analysis tools. Marinho et al. [20] proposed an algorithm to compute an upper-bound on the CRPD for a task executing using preemption triggered floating non-preemptive regions. Additionally, Peng et al. [21] proposed a pseudo-polynomial preemption point placement algorithm for control flow graphs with arbitrarily nested conditional program structures. The CRPD cost model used in these papers does not take into account the interdependency between selected preemption points.

Our work improves these results by computing the cache related preemption delay (CRPD) contribution to the variable preemption overhead cost as a function of the current and next selected preemption points. Our method not only accounts for the cache blocks evicted due to preemption, but also accounts for the cache blocks that are reloaded during execution between preemption points thereby improving the accuracy of marginal CRPD computations for multiple nested preemptions. Due to the variability in CRPD and preemption overhead cost, we propose a dynamic programming algorithm to realize a globally optimal preemption point placement that further minimizes the number of preemptions and the preemption overhead cost due to the increased CRPD accuracy.

III. System Model

Our system contains a task set τ of n periodic or sporadic tasks (τ_1, τ_2, ..., τ_j, ..., τ_n) each scheduled on a single processor. Each task τ_i is characterized by a tuple (C_i^{NP}, D_i, T_i) where C_i^{NP} is the non-preemptive worst case execution time, D_i is the relative deadline, and T_i is the inter-arrival time or period. Each task τ_i creates an infinite number of jobs, with the first job arriving at any time after system start time and subsequent jobs arriving no earlier than T_i time units with a relative deadline $D_i \leq T_i$. The system utilizes a preemptive scheduler with each task/job containing N_i number of basic blocks denoted (δ_i^0, δ_i^1, δ_i^2, ..., $\delta_i^{N_i}$) with the total number of basic blocks given by N = $\Sigma_i N_i$. A dummy basic block δ_i^0 with zero WCET is added at the beginning of each task to capture the preemption that occurs prior to task execution. In our sequential model, a basic block is a set of one or more instructions that execute non-preemptively. Basic blocks are essentially the vertices V of a linear control flow graph (CFG) connected in a sequence by edges E representing the execution sequence of one or more job instructions. Figure 1 shown below illustrates the sequential basic block connection structure. Preemptions are

b_i^0	ξ_i^0	b_i^1	ξ_i^1	b_i^2	ξ_i^2	b_i^3	ξ_i^3	b_i^4	ξ_i^4	b_i^5	ξ_i^5	b_i^6	ξ_i^6
δ_i^0		δ_i^1		δ_i^2		δ_i^3		δ_i^4		δ_i^5		δ_i^6	

Fig. 1: System Model.

permitted at basic block boundaries. We introduce the basic block notation δ_i^j where i is the task identifier and j is the basic block identifier. We introduce the the non-preemptive basic block execution time notation b_i^j where i is the task identifier and j is the basic block identifier, hence using this convention we have

$$C_i^{NP} = \Sigma_j \; b_i^j. \tag{1}$$

The processor utilization U_i of task τ_i is given by

$$U_i = C_i^{NP}/T_i. \tag{2}$$

The maximum blocking time β_i is the maximum duration that a task τ_i can be blocked due to non-preemptive execution of a lower priority task. The preemption cost as a function of the current and next preemption points is given by the variable ξ_i as discussed in section V. Tasks may be preempted by multiple higher-priority tasks identified by the variable k. For example for both Deadline-Monotonic (DM) and Earliest-Deadline-First (EDF) scheduling, the set of higher priority tasks is represented by:

$$k \in hp(i) = \{k | D_k < D_i\}. \tag{3}$$

The number of times task τ_k can preempt task τ_i during task τ_i's execution is given by the term

$$N_p(\tau_i, \tau_k) = \lceil (C_i/T_k) \rceil \geq 1. \tag{4}$$

In a limited preemption approach, each task is permitted to execute non-preemptively for a maximum amount of time denoted by Q_i. Previous research on limited preemption scheduling (e.g., Baruah [15]) has used the above information to determine the value of Q_i for each task. Therefore, we assume that Q_i is provided by such an approach.

IV. CRPD COMPUTATION

While existing research has focused on computing the upper bounds on cache related preemption delay (CRPD), our approach achieves higher accuracy by computing the *loaded cache blocks* (LCBs), as defined below, due to higher priority task preemption by using the set of potential/chosen preemption locations. The sets of various cache blocks are represented as sets of integers. We employ the CRPD terms UCB and ECB as defined by Lee et al. [7] and by Altmeyer and Burguiere [9].

Definition 1. Useful Cache Block (UCB): A memory block m is called a useful cache block at program point $\delta_i^{j_1}$, if
(a) m may be cached at $\delta_i^{j_1}$ and
(b) m may be reused at program point $\delta_i^{j_2}$ that may be reached from $\delta_i^{j_1}$ without eviction of m on this path.

More formally, cache block $m \in \mathit{UCB}(\tau_i)$ if and only if τ_i has m as a useful cache block in some cache-set s. Note this definition of *UCB* embodies a task level view. Cache block $m \in \mathit{UCB}_{\text{out}}(\delta_i^j)$ if and only if δ_i^j has m as a useful cache block in some cache-set s where

$$\mathit{UCB}(\tau_i) = \bigcup_{\delta_i^j \in \tau_i} \mathit{UCB}_{\text{out}}(\delta_i^j). \tag{5}$$

The notation $\mathit{UCB}_{\text{out}}(\delta_i^j)$ is the set of useful cache blocks cached in task τ_i post-execution of basic block δ_i^j. Similarly, the notation $\mathit{UCB}_{in}(\delta_i^j)$ is the set of useful cache blocks cached in task τ_i pre-execution of basic block δ_i^j.

Definition 2. Evicting cache block (ECB): A memory block m of the preempting task is called an evicting cache block, if it may be accessed during the execution of the preempting task.

Cache block $m \in \mathit{ECB}(\tau_k)$ if and only if τ_k may evict m in some cache-set s. Note this definition of *ECB* also embodies a task level view. Cache block $m \in \mathit{ECB}(\delta_i^j)$ if and only if δ_k^j may evict m in some cache-set s where

$$\mathit{ECB}(\tau_k) = \bigcup_{\delta_k^j \in \tau_k} \mathit{ECB}(\delta_k^j). \tag{6}$$

The notation $\mathit{ECB}(\delta_k^j)$ is the set of evicting cache blocks accessed in task τ_k during execution of basic block δ_k^j. In order to determine which cache blocks may be reloaded once preemption occurs, we introduce the notion of an accessed useful cache block (AUCB).

Definition 3. Accessed useful cache block (AUCB): A memory block of the preempted task is called an accessed useful cache block if it may be accessed during the execution of a basic block δ_i^j for task τ_i.

The term $\mathit{AUCB}_{\text{out}}(\delta_i^j)$ represents the useful cache blocks (UCBs) accessed by task τ_i during execution of basic block at location δ_i^j. The definition of *AUCB* is introduced to capture the set of task accessed memory at a specific basic block location subsequently used in the calculation of blocks that must be reloaded when task preemptions occur. We compute the set of accessed useful cache blocks (AUCBs) as follows:

$$\mathit{AUCB}_{\text{out}}(\delta_i^j) = \mathit{UCB}_{\text{out}}(\delta_i^j) \cap \mathit{ECB}(\delta_i^j). \tag{7}$$

where $\mathit{UCB}_{\text{out}}(\delta_i^j)$ is the set of useful cache blocks for task τ_i post basic block δ_i^j execution; and $\mathit{ECB}(\delta_i^j)$ denotes the set of cache blocks accessed in task τ_i during execution of basic block δ_i^j. It is important to note that only cache block evictions due to preemption are considered, as intrinsic cache misses are captured as part of WCET analysis in the term C_i^{NP}. Using the previously defined terms, we may now define and explicitly compute the cache blocks that are re-loaded due to preemption which are called loaded cache blocks (LCBs).

Definition 4. Loaded cache block (LCB): A memory block of the preempted task is called a loaded cache block, if it may be re-loaded during the non-preemptive region (i.e., within a series of basic blocks with no preemptions) immediately following a preemption.

$\mathit{LCB}(\delta_i^{curr}, \delta_i^{next})$ denotes the set of cache blocks re-loaded during execution of the non-preemptive region between basic block δ_i^{curr} and basic block δ_i^{next}, resulting from preemption of task τ_i, where basic block location δ_i^{curr} and δ_i^{next} are the potential/selected preemption point and next potential/selected preemption point respectively. In our model, a preemption point located at basic block δ_i^{curr} occurs at the edge between δ_i^{curr} and δ_i^{curr+1}.

The definition of *LCB* is introduced to capture the set of reloaded cache memory at a specific basic block as a function of the current and next selected preemption points. Here, we account for the overhead within a non-preemptive region for reloading UCBs that could have potentially been evicted by the preemption occurring immediately after δ_i^{curr} and used by some basic block prior to the preemption occurring immediately after δ_i^{next}.

$$\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}) = [\mathit{UCB}_{\text{out}}(\delta_i^{curr}) \cap [\cup_{\nu \in \lambda} \mathit{AUCB}_{\text{out}}(\delta_i^{\nu})]] \cap [\cup_{\tau_k \in hp(i)} \mathit{ECB}(\tau_k)] \tag{8}$$

$$\text{where } \lambda \stackrel{\text{def}}{=} \{\nu | \nu \in [\mathit{curr}{+}1, \mathit{curr}{+}2, \ldots, \mathit{next}]\}$$

δ_i^{curr} represents the current selected preemption point where $\delta_i^{curr} \in \rho_i$ and δ_i^{next} represents the next selected preemption point where $\delta_i^{next} \in \rho_i$, and $\rho_i \subseteq \{\delta_i^0, \delta_i^1, \ldots, \delta_i^{N_i}\}$ is an ordered set by ascending index of selected preemption points for task τ_i:

$$\rho_i \stackrel{\text{def}}{=} \{\delta_i^m | \delta_i^m \text{ is a selected preemption point of task } \tau_i \wedge m \in [0, 1, 2, \ldots, N_i]\}$$

This formula for $\mathit{LCB}(\delta_i^{curr}, \delta_i^{next})$ results in the accounting

of loaded cache blocks where the preemption occurs. Note that this notation assumes a sequential basic block structure. Once we have the set of cache blocks that must be re-loaded due to preemption, the CRPD related preemption overhead may be computed as shown below.

$$\gamma_i(\delta_i^{curr}, \delta_i^{next}) = |LCB(\delta_i^{curr}, \delta_i^{next})| \cdot BRT. \quad (9)$$

where BRT is the cache block reload time; and $LCB(\delta_i^{curr}, \delta_i^{next})$ represents the loaded cache blocks or memory accessed by the preempted task τ_i at basic block δ_i^{curr} caused by higher priority preempting tasks. To illustrate our approach for computing LCBs, a suitable example is presented in the appendix due to space limitations here.

V. Integrated WCET/CRPD Calculation

The modified preemption cost as a function of the current and next preemption points is given by:

$$\begin{aligned}\xi_i(\delta_i^{curr}, \delta_i^{next}) \ &= \gamma_i(\delta_i^{curr}, \delta_i^{next}) + \pi + \sigma + \\ &\quad \eta(\gamma_i(\delta_i^{curr}, \delta_i^{next})).\end{aligned} \quad (10)$$

where π is the pipeline cost, σ is the scheduler processing cost, and $\eta()$ is the front side bus contention resulting from the cache reload interference as described in [1]–[3]. The output of our algorithm is an optimal set of preemption points, with respect to our computed preemption cost $\xi_i()$, subject to the maximum allowable non-preemption region Q_i. The optimal set of preemption points obtained using the enhanced accuracy of our preemption cost computation is used to calculate each task's WCET with preemption overhead given by:

$$C_i = B_i(\rho_i) = C_i^{NP} + \sum_{m=1}^{|\rho_i|-1} [\xi_i(\rho_i^m, \rho_i^{m+1})] \quad (11)$$

where ρ_i^m is the m^{th} selected preemption point for task τ_i. To further clarify what we mean by preemption, we say that δ_i^m is a "preemption point" if the preemption between basic blocks δ_i^m and δ_i^{m+1} is enabled, meaning the scheduler may preempt between these two basic blocks. Commensurate with our preemption point placement algorithm discussed later, preemptions are always taken at basic blocks δ_i^0 and $\delta_i^{N_i}$ hence $\delta_i^0,\ \delta_i^{N_i} \in \rho_i$ for any feasible set of preemption points ρ_i. Supplemental clarification of this requirement is shown in the example of LCB interdependence in the appendix. The complete problem formulation with constraints for finding the minimum WCET with preemption overhead cost $B_i(\rho_i)$ for task τ_i is given by:

$$\begin{aligned}B_i(\rho_i) = \min_{\rho_i \in \tau_i} \Big\{ \Big[\sum_{m=1}^{|\rho_i|-1} \xi_i(\rho_i^m, \rho_i^{m+1}) + \sum_{s=1}^{N_i} b_i^s \Big] \ | \\ \Psi_i(\rho_i) \ = \ True \Big\}\end{aligned} \quad (12)$$

The selection of optimal preemption points is subject to the constraint $\Psi_i(\rho_i)$ that no non-preemptive region in task τ_i exceeds the maximum allowable non-preemption region parameter Q_i:

$$\Psi_i(\rho_i) = \left\{ \begin{array}{ll} \text{True}, & \text{if } q_i^m(\rho_i) \le Q_i \text{ for } m \in [1, |\rho_i| - 1] \\ \text{False}, & \text{otherwise} \end{array} \right\} \quad (13)$$

where $q_i^m(\rho_i)$ represents the m^{th} non-preemptive-region (NPR) time for task τ_i, capturing the cost of the preemption ρ_i^m given that ρ_i^{m+1} is the next selected preemption point, plus the basic block cost of all blocks between ρ_i^m and ρ_i^{m+1}:

$$q_i^m(\rho_i) = \Big[\xi_i(\rho_i^m, \rho_i^{m+1}) + \sum_{s:\delta_i^s \in \{\rho_i^m, \ldots, \rho_i^{m+1}\}} b_i^s \Big] \quad (14)$$

Our approach employs the results of schedulability analysis and the aforementioned WCET + CRPD calculation with the maximum allowable non-preemption region parameter Q_i computed for each task τ_i. The objective is to select a subset of preemption points that minimizes each tasks WCET + CRPD parameter C_i. We introduce slight variations of terms Ψ_i, q_i, and B_i, used in solving intermediate subproblems of our proposed algorithm. The selection of optimal preemption points is subject to the constraint that no non-preemptive region in task τ_i exceeds the maximum allowable non-preemption region parameter Q_i:

$$\Psi_i(\delta_i^j, \delta_i^k) = \left\{ \begin{array}{ll} \text{True}, & \text{if } q_i(\delta_i^j, \delta_i^k) \le Q_i \text{ for } \delta_i^j, \delta_i^k \in \rho_i \\ \text{False}, & \text{otherwise} \end{array} \right\} \quad (15)$$

where $q_i(\delta_i^j, \delta_i^k)$ represents a possible candidate optimal non-preemptive-region (NPR) time with successive preemption points at basic block locations δ_i^j and δ_i^k for task τ_i:

$$q_i(\delta_i^j, \delta_i^k) = \Big[\xi_i(\delta_i^j, \delta_i^k) + \sum_{n=j+1}^{k} b_i^n \Big] \quad (16)$$

The next expression gives a recursive solution to the CRPD+WCET minimization problem for the subproblem of the first i basic blocks. We compute the WCET including the preemption point δ_i^k using the term $B_i(\delta_i^k)$ for task τ_i as given by:

$$\begin{aligned}B_i(\delta_i^k) = \min_{m \in \{0,1,\ldots,k-1\}} \Big\{ \Big[B_i(\delta_i^m) + q_i(\delta_i^m, \delta_i^k) \Big] \ | \\ \Psi_i(\delta_i^m, \delta_i^k) \ = \ True \Big\}\end{aligned} \quad (17)$$

when $k \in \{1, 2, \ldots, N_i\}$. For the base case where k = 0, we have $B_i(\delta_i^0) = 0$. This recursive formulation is necessary for developing the dynamic programming solution presented in the next section. The following theorem shows our problem has optimal substructure, and thus the formulation of Equation 17 represents an optimal solution.

Theorem 1. *The WCET + CRPD cost variable $B_i(\delta_i^k)$ utilized in Equation 17 exhibits optimal substructure.*

Proof: Let Δ_i^k be the subproblem of the sequential

control flowgraph containing basic blocks $\{\delta_i^0, \delta_i^1, \ldots, \delta_i^k\}$. To prove optimal substructure, we show that we can obtain the optimal set of preemption points for minimizing the WCET+CRPD for any Δ_i^k by using the optimal solutions to subproblems $\Delta_i^0, \Delta_i^1, \ldots, \Delta_i^{k-1}$. Let $B_i(\delta_i^0), B_i(\delta_i^1), \ldots, B_i(\delta_i^{k-1})$ represent the cost of the optimal solution to these subproblems. We need to show that Equation 17 represents the optimal cost to Δ_i^k.

By way of contradiction, assume there is a better feasible solution ρ_i' for the sub-problem of determining the optimal limited preemption execution costs from basic block δ_i^0 to basic block δ_i^k; that is, $B_i(\rho_i')$ is strictly smaller than the solution to Δ_i^k obtained in Equation 17 (i.e., $B_i(\delta_i^k)$). Let δ_i^ℓ (where $\ell \in \{0, 1, \ldots, k-1\}$) be the last preemption point prior to δ_i^k in the set ρ_i', and let ρ_i'' be the set of preemption points obtained from ρ_i' by removing the preemption point after δ_i^k (i.e., ρ_i'' is a solution to Δ_i^ℓ). Thus, we can represent the cost of the solution ρ_i' (i.e., $B_i(\rho_i')$ by the left-hand-side of the following inequality:

$$B_i(\rho_i'') + q_i(\delta_i^\ell, \delta_i^k) < B_i(\delta_i^k). \tag{18}$$

Since ρ_i' is a feasible solution to Δ_i^k, it must be that $\Psi_i(\delta_i^\ell, \delta_i^k)$ is true. Hence, from Equation 17 and the min operation, we can obtain an upper bound on $B_i(\delta_i^k)$ by considering the solution to subproblem Δ_i^ℓ:

$$B_i(\delta_i^k) \leq B_i(\delta_i^\ell) + q_i(\delta_i^\ell, \delta_i^k). \tag{19}$$

Combining the inequalities of Equations 18 and 19, we finally obtain $B_i(\rho_i'') < B_i(\delta_i^\ell)$. However, this contradicts our assumption at the beginning of the proof that $B_i(\delta_i^\ell)$ represented an optimal solution to subproblem Δ_i^ℓ. Thus, a solution ρ_i' with smaller cost than $B_i(\delta_i^k)$ cannot exist, and Equation 17 computes the minimum obtainable cost for the problem Δ_i^k. ■

VI. Preemption Point Placement Algorithm

Implementing a recursive algorithm directly from Equation 17 would lead to a computationally intractable implementation. Instead, we now propose an $O(N_i^2)$ dynamic programming algorithm for computing the optimal preemption points. Our dynamic programming preemption point placement algorithm is summarized in Algorithm 1 shown below. For each task τ_i, we are given the following parameters: 1) the number of basic blocks N_i , 2) the non-preemptive execution time of each basic block b_i, 3) the maximum allowable non-preemptive region Q_i, and 4) the preemption cost matrix ξ_i. The preemption cost matrix ξ_i is organized for each basic block δ_i^j and contains the preemption cost for all successor basic blocks of the task's control flow graph. The minimum preemption cost up to all basic blocks is computed and stored in an array denoted B_i. Each entry of the B_i array is initialized to infinity. As we consider whether each basic block δ_i^k is in the set of optimal preemption points, the location of the previous basic block δ_i^j with minimal preemption cost is stored in an array denoted ρ_{prev}. The algorithm examines each basic block from δ_i^1 to $\delta_i^{N_i}$ to minimize the preemption cost by traversing backwards from the current basic block δ_i^k under consideration in order to find the basic block δ_i^j with minimal preemption cost subject to the constraint $q_i(\delta_i^j, \delta_i^k) \leq Q_i$. While each basic block will have a predecessor with minimum preemption cost, the list of selected preemption points is obtained by starting with basic block $\delta_i^{N_i}$ and hopping to the predecessor basic block stored at $\rho_{prev}(\delta_i^{N_i})$, denoted δ_i^m. Basic blocks $\delta_i^{N_i}$ and δ_i^m are added to the optimal preemption point set ρ_i. This basic block hopping process continues until basic block δ_i^0 is reached. The set ρ_i contains the complete list of selected optimal preemption points.

Algorithm 1 D.P. Optimal Preemption Point Placement

```
1:  function Select_Optimal_PPP(N_i,b_i,Q_i,ξ_i)
2:      B_i ← ∞ ρ_prev ← {δ_i^0};
3:      if b_i^k > Q_i for some k ∈ {1,...,N_i} then
4:          return INFEASIBLE;
5:      end if
6:      B_i(δ_i^0) ← 0;
7:      for k : 0 ≤ k ≤ N_i do
8:          C_i^NP(δ_i^k,δ_i^k) ← 0;
9:          q_i(δ_i^k, δ_i^k) ← 0;
10:         for j : k − 1 ≥ j ≥ 0 do
11:             C_i^NP(δ_i^j, δ_i^k) ← b_i^{j+1} + C_i^NP(δ_i^{j+1}, δ_i^k);
12:             q_i(δ_i^j, δ_i^k) ← ξ_i(δ_i^j, δ_i^k) + C_i^NP(δ_i^j, δ_i^k);
13:             if q_i(δ_i^j, δ_i^k) ≤ Q_i then
14:                 P_cost ← B_i(δ_i^j) + q_i(δ_i^j, δ_i^k);
15:                 if P_cost < B_i(δ_i^k) then
16:                     B_i(δ_i^k) ← P_cost;
17:                     ρ_prev(δ_i^k) ← δ_i^j;
18:                 end if
19:             end if
20:         end for
21:     end for
22:     ρ_i ← Compute_PPSet(N_i, ρ_prev);
23:     return FEASIBLE;
24: end function
25:
26:
27: function Compute_PPSet(N_i,ρ_prev)
28:     Computes ρ_i from ρ_prev (Details omitted)
29: end function
```

To exemplify how our algorithm works, consider the following example. Let $N_i = 6$ and $Q_i = 12$ for the following basic block structure with WCET and preemption costs shown in Figure 2. The algorithm computes and stores the cumulative non-preemptive execution costs for starting and ending basic block pairs in a matrix denoted $C_i^{NP}(\delta_i^j,\delta_i^k)$. For example, $C_i^{NP}(\delta_i^1,\delta_i^3) = b_i^2 + b_i^3 = 2 + 2 = 4$. We don't include b_i^1 since the preemption occurs after execution of basic block δ_i^1. Using this information, the combined WCET and CRPD costs for each basic block pair is computed and stored in a matrix denoted q_i. For example, $q_i(\delta_i^1,\delta_i^3) = C_i^{NP}(\delta_i^1,\delta_i^3) + \xi_i(\delta_i^1,\delta_i^3) = 4 + 5 = 9$.

0	-	3	-	2	-	2	-	3	-	3	-	3	-
δ_i^0	ξ_i^0	δ_i^1	ξ_i^1	δ_i^2	ξ_i^2	δ_i^3	ξ_i^3	δ_i^4	ξ_i^4	δ_i^5	ξ_i^5	δ_i^6	ξ_i^6

ξ_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
δ_i^0	**0**	1	2	4	4	3	2
δ_i^1	-	**0**	3	5	6	4	3
δ_i^2	-	-	**0**	8	7	5	4
δ_i^3	-	-	-	**0**	8	7	6
δ_i^4	-	-	-	-	**0**	6	7
δ_i^5	-	-	-	-	-	**0**	8
δ_i^6	-	-	-	-	-	-	**0**

Fig. 2: Algorithm Example.

The remaining q_i matrix entries are computed in a similar fashion. These matrices are shown in Figure 3. The shaded cells in the q_i matrix represent cases where the combined WCET and CRPD costs for these basic block pairs exceed the maximum allowable non-preemptive region parameter Q_i. During execution of the algorithm, the minimum combined WCET + CRPD costs are computed for each basic block and stored in an array denoted B_i. Basic block pairs with preemptive costs that are less than or equal to Q_i are candidates for selection. When a lower cost is determined for a given basic block, the predecessor preemption point is updated in the ρ_{prev} array which keeps track of the selected predecessor preemption points thereby forming a daisy chain containing the entire set of selected preemption points. The final results are illustrated in Figures 3 and 4 below.

q_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
δ_i^0	**0**	4	7	11	14	17	19
δ_i^1	-	**0**	5	9	13	14	16
δ_i^2	-	-	**0**	10	12	13	15
δ_i^3	-	-	-	**0**	11	13	15
δ_i^4	-	-	-	-	**0**	9	13
δ_i^5	-	-	-	-	-	**0**	11
δ_i^6	-	-	-	-	-	-	**0**

Fig. 3: Combined WCET and CPRD Costs.

B_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
δ_i^0	-	4	7	11	19	28	**39**

ρ_{PREV}	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^2}$	$\boldsymbol{\delta_i^4}$	$\boldsymbol{\delta_i^5}$

ρ_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
	$\boldsymbol{\delta_i^0}$		$\boldsymbol{\delta_i^2}$		$\boldsymbol{\delta_i^4}$	$\boldsymbol{\delta_i^5}$	$\boldsymbol{\delta_i^6}$

0	2	3	0	2	7	2	0	3	6	3	8	3	0
δ_i^0	ξ_i^0	δ_i^1	ξ_i^1	δ_i^2	ξ_i^2	δ_i^3	ξ_i^3	δ_i^4	ξ_i^4	δ_i^5	ξ_i^5	δ_i^6	ξ_i^6

Fig. 4: Algorithm Results.

The maximum blocking time Q_i that each task may tolerate utilizes the computed task WCET parameter C_i. The method for obtaining the maximum blocking time is eloquently summarized for the Earliest Deadline First (EDF) and Fixed Priority (FP) scheduling by Bertogna et al. algorithms [4] [19]. The circular dependency between the maximum blocking time Q_i and task WCET C_i parameters suggests an iterative approach to allow the two parameters to convergence to a steady state. One such iterative approach contains the following steps as given in Algorithm 2. Convergence of Algorithm 2 results primarily from the fact that the optimal preemption points don't change unless the parameter Q_i changes. Since Q_i represents the execution time slack available to a task τ_i, as expected Q_i has been observed to exhibit non-increasing values during each subsequent algorithm iteration.

Algorithm 2 Iterative Schedulability and Preemption Point Placement Algorithm

1: Assume the CRPD of the task system is initially zero.
2: **repeat**
3: Run the Baruah algorithm to obtain the maximum non-preemptive region Q_i for each task.
4: Select optimal preemption points using our dynamic programming algorithm and CRPD calculation.
5: Compute the WCET + CRPD C_i from the selected preemption points.
6: **until** the selected preemption points do not change or the system is not feasible for the computed WCET + CRPD.

VII. Evaluation

The evaluation of our preemption point placement algorithm will embody two methods: 1) characterization and measurement of preemption costs using real-time application code, and 2) a breakdown utilization schedulability comparison for various CRPD computational approaches.

A. Preemption Cost Characterization

To characterize the behavior and estimate the benefit of the approach proposed in this paper, a case study of representative tasks was performed. The tasks were randomly selected from the Malardalen University (MRTC) WCET benchmark suite [22]. Each task was built using Gaisler's Bare-C Cross Compiler [23] for the GRSIM LEON3 [24] simulated target.

Tasks were first analyzed using AbsInt's a^3 WCET [25] to determine the set of basic blocks. Next, the basic blocks $\{\delta_i^0, \delta_i^1, \delta_i^2, ..., \delta_i^{N_i}\}$ were serialized by recording their order during execution. Program points were identified as the address of the final instruction of each basic block δ_i^j for $j \in [0, N_i]$ to match the sequential basic block structure used by our preemption point placement algorithm. Each program point served as a breakpoint when running the task on the simulator. In accordance with the sequential basic block structure used in our algorithm, only the data cache is used to compute CRPD as the linear basic block structure offers only limited opportunities for instruction cache blocks to be revisited.

During the execution of each basic block, the data cache states were saved as $\Upsilon_D(\delta_i^j)$ at each breakpoint. The cache snapshots were selected as the final visit of each program point where the data cache contents $\Upsilon_D(\delta_i^j)$ were captured, and recorded. From the final $\Upsilon_D(\delta_i^j)$ snapshots, a conservative estimate of LCBs shared between program points was determined.

Shared LCBs were calculated by intersecting the cache state snapshots from δ_i^j to δ_i^k, except δ_i^k. A cache line that remains unchanged after the execution of $\{\delta_i^{j+1}, \delta_i^{j+2}, ..., \delta_i^k\}$ will be present in the cache before execution of the basic block that δ_i^k represents. It is only from these unchanged cache lines that the shared LCBs between δ_i^j and δ_i^k can be selected. The complete set of unchanged cache lines serves as a safe upper-bound on the LCBs shared between each basic block pair. The equation below formalizes this idea, using the data cache snapshots $\Upsilon_D(\delta_i^j)$.

$$LCB(\delta_i^j, \delta_i^k) \subseteq \bigcap_{m=j+1}^{k} \Upsilon_D(\delta_i^m) \tag{20}$$

1) Availability: This method may be verified and reproduced using the same tools and data. Gaisler's compiler and simulator are freely available. AbsInt's a^3 tool is available for educational and evaluation purposes. The programs written and data used in this paper can be found on GitHub [26] thereby permitting the research community to reproduce and leverage our work as needed.

2) Results: The results are presented as an illustration of the potential benefit of our proposed method, utilizing pairs of preemptions to determine costs, over methods that consider only the maximum CRPD at a particular preemption point (e.g., Bertogna et al. [4]). In terms of LCB computation this implies that the maximum LCB value over all subsequent program points must be used as the CRPD cost:

$$\max\{LCB(\delta_i^j, \delta_i^k) \mid j < k\} \tag{21}$$

In the following graphs, each point in the graph represents two points in the program. The first point of the program δ_i^j is fixed by the x-axis. The y-axis indicates the shared LCB count with a later program point. The first graph shown in Figure 5 is for the recursion program of the MRTC benchmark suite. The lines represent the minimum and maximum shared LCBs between various program points. At each point on the graph, the index of the next preemption point associated with the respective minimum or maximum graph value at that location is shown. At program point δ_i^4, the minimum CRPD value is coupled with program point δ_i^7 having a shared LCB count of 14 whereas the single-valued CRPD computation method finds 24 shared LCBs coupled at program point δ_i^5.

To compare the potential utility of a method that uses at each preemption a single-valued CRPD and our method which determines CRPD based on a pair of adjacent preemption points, consider any program point on the horizontal access of Figures 5, 6, or 7. For any program preemption point δ_i^k, a single-valued approach would, to be safe, use the value reported in the larger-valued dashed line. However, an approach that considered pairs of preemptions, as in our approach, can reduce the value and potentially obtain a CRPD reported on the solid line. The difference between the performance of these two preemption point placement algorithms is an example of the benefit provided by considering location aware CRPD cost.

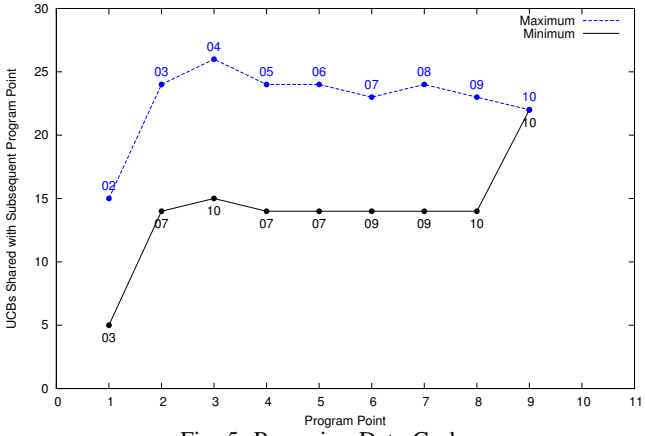

Fig. 5: Recursion Data Cache.

The second graph represents the lms benchmark task data cache shown in Figure 6, and the third graph represents the adpcm benchmark task data cache shown in Figure 7. Similar to the recursion benchmark data cache, the variability in the minimum and maximum shared LCBs for each program point further exemplifies the benefit of using location aware CRPD cost in preemption point placement. Maximum and minimum data cache costs for the other seven tasks show similar variability but are not shown here due to space limitations. In summary, the CRPD cost was consistently reduced or maintained compared to the single-valued approach. A maximum of 68% reduction of contributing cache lines in the bsort benchmark along with an average of 18.6% decrease over all benchmarks was notably observed. The ability of our method to leverage this reduction leads to the schedulability improvements observed in the next subsection.

In examining the graphs for all tasks and data caches a few common traits were noted. The minimum shared LCBs sharply increases at the end of each task's execution. This is due to the nature of the conservative estimation of shared LCBs and the tasks, and the number of instructions in the basic block between the final program points is relatively small thereby limiting the number of data cache lines that could be eliminated by the intersection.

Drastic spikes downwards in the shared LCB counts for the minimum and maximum curves coincide with function call boundaries, or large conditional blocks. At these boundaries, the maximum and minimum LCB counts are approximately the same. There is a sharp upward spike in the early program

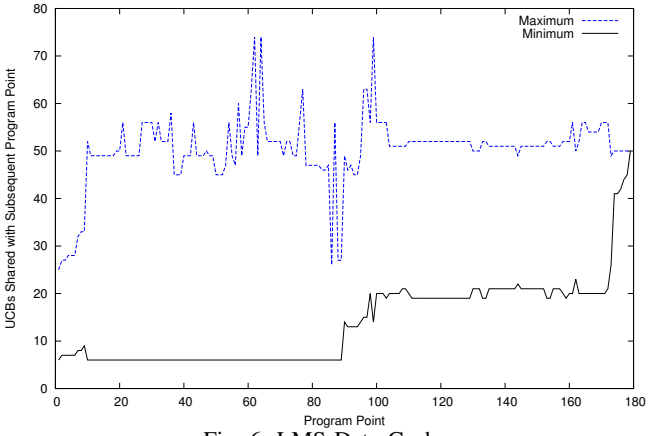

Fig. 6: LMS Data Cache.

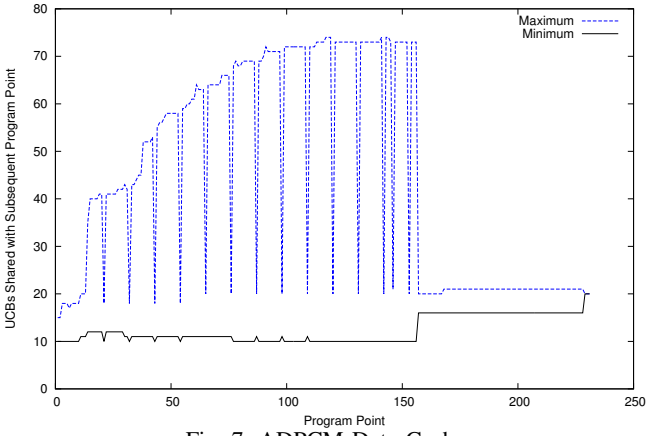

Fig. 7: ADPCM Data Cache.

points for the maximum curves. This trend is due to the early initialization blocks built into tasks. In our analysis, the minimum curves show a clear benefit of selecting preemption points outside of the early initialization section.

B. Breakdown Utilization

The previous analysis illustrates the benefit of using a more precise CRPD cost in preemption point placement thereby reducing the preemption overhead for an individual task, however, it does not address the benefits to task set schedulability. To evaluate task set schedulability benefits a second case study was performed focusing on the breakdown utilization of the MRTC WCET benchmark suite [22] for various algorithms. Our study compares the Luniss et al. UCB only approach for EDF [27], the Bertogna et al. Explicit Preemption Point Placement algorithm [4], and our improved Explicit Preemption Point Placement algorithm. For notational convenience the UCB Only approach for EDF will be referred to as UOE, the Bertogna Explicit Preemption Point Placement algorithm as $BEPP$, and our Explicit Preemption Point Placement algorithm as EPP.

The appropriate schedulablity test for UOE is comprised of three parts: $\gamma_{t,j}^{ucb}$, U_j^*, and U^* each representing the maximum CRPD for task τ_j, the utilization of task τ_j including CRPD, and the utilization of the task set respectively as documented in Lunniss *et al.* [27]. A task set is schedulable when $U^* \leq 1$.

Borrowing the breakdown utilization evaluation technique from [27], each task has its deadline and period set to $T_i = P_i = u \cdot C_i$ where u is a constant. The constant, u, begins at the number of tasks (ten) and is increased in steps of 0.25 until the task set becomes schedulable. Incremental negative adjustments are then made to determine when the set becomes unschedulable, indicating the final breakdown utilization. For each task, the set of shared LCBs are calculated at each program point. Taking the maximum shared LCB count for any task is safe and appropriate for calculating U^*.

For $BEPP$, the maximum shared LCB counts obtained in the earlier evaluation serve as input. Lastly, for EPP, the shared LCB counts obtained in the earlier evaluation serve as input for our Explicit Preemption Point Placement algorithm. The last input variables required for both approaches are C_i and BRT. C_i was captured as the total number of cycles required to complete the task without preemptions. The breakdown utilization study sweeps the BRT parameter from 10 μs to 390 μs, representing values across several different types of processors.

The breakdown utilization determination leverages our iterative schedulability and preemption point placement algorithm as outlined in the following steps below as given in Algorithm 3. Using ten tasks, the breakdown utilization comparison between UOE, $BEPP$, and EPP are summarized in Figure 8. The breakdown utilization results indicate that $BEPP$ dominates the UOE algorithm primarily due to the limited preemption model utilizing CRPD cost at the basic block level. EPP dominates $BEPP$ resulting from the more accurate location aware CRPD cost used in our preemption point placement algorithm. As expected, the breakdown utilization converges for all three methods for small BRT values as the cache-overhead becomes negligible.

Algorithm 3 Breakdown Utilization Evaluation Algorithm

```
1: Start with a task system that may or may not be feasible.
2: Assume the CRPD of the task system is initially zero.
3: repeat
4:     Run the Iterative Schedulability and Preemption Point Placement Algorithm 2
5:     if the task system is feasible/schedulable then
6:         Increase the system utilization by decreasing the periods via a binary search.
7:     else
8:         Decrease the system utilization by increasing the periods via a binary search.
9:     end if
10: until the utilization change is less than some tolerance.
11: The breakdown utilization is given by U.
```

VIII. Conclusion

In this paper, we presented an enhanced approach for calculating the CRPD taking into account the selected pre-

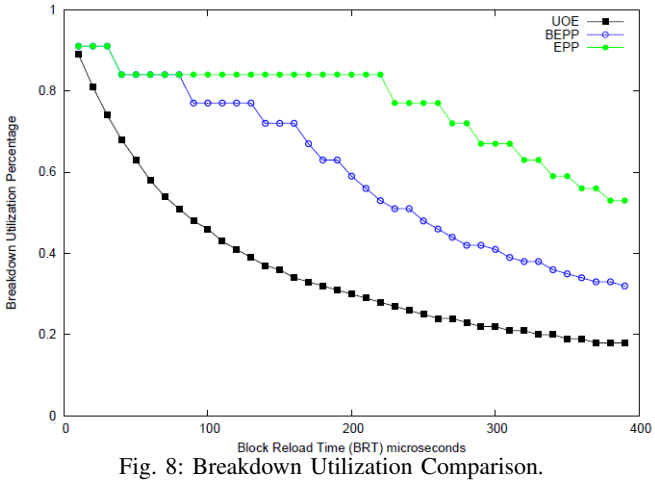

Fig. 8: Breakdown Utilization Comparison.

emption points resulting in greater accuracy. Using a more precise CRPD calculation, we also presented an improved algorithm for selecting a limited number of preemption points for each task subject to schedulability constraints. Our improved preemption placement algorithm was demonstrated to minimize the overall preemption cost, an important result in achieving schedulability in real-time systems. We highlighted the iterative nature of considering schedulability constraints in our preemption point placement algorithm and proposed an algorithm combining schedulability analysis with limited preemption point placement. This approach effectively illustrates how the individual tasks non-preemption region parameters Q_i and the optimal selected preemption points will eventually converge. Furthermore, our enhanced algorithm was demonstrated to be optimal in that if a feasible schedule is not found, then no feasible schedule exists by any known method utilizing a static Q_i value. Our algorithm was shown to run in quadratic time complexity. Potential preemption points can be defined automatically using Gaisler's compiler and simulator along with AbsInt's a^3 tool or defined manually by the programmer during design and implementation. Our experiments demonstrated the effectiveness of the enhanced CRPD calculation by illustrating the benefits using the task set from the MRTC WCET benchmark suite [22]. We also demonstrated the benefits of our enhanced limited optimal preemption point placement algorithm and its increased system schedulability as compared to other algorithms. While our task model is defined using a linear sequence of basic blocks, it was deemed a highly suitable model to introduce our revised methods for enhanced CRPD calculation and optimal limited preemption point placement. The need to subsume higher level programming constructs being the prominent assumption of the linear basic block structure can potentially diminish the utility of our approach if the non-preemptive execution time of any basic block violates the constraint $C_i^{NP} > Q_i$. For this case, we need to "break-up" the basic block by permitting preemption every Q_i time units.

In future work, we plan to 1) extend the techniques described here to set-associative caches, 2) perform a schedulability comparison of synthetic task set for various preemption models, and 3) remove the linear basic block restriction thereby permitting arbitrary control flowgraphs.

References

[1] R. Pellizzoni and M. Caccamo, "Toward the predictable integration of real-time cots based systems," *In RTSS, 2007.*

[2] R. Pellizzoni, B. Bui, M. Caccamo, and L. Sha, "Coscheduling of cpu and i/o transactions in cots-based embedded systems," *In RTSS, 2008.*

[3] R. Pellizzoni, E. Betti, S. Bak, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," *In RTAS, 2011.*

[4] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," *In Proceedings ECRTS, 2011, IEEE.*

[5] C.-G. Lee, J. Hahn, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *In RTSS, 1996.*

[6] ——, "Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *In RTSS, 1997.*

[7] ——, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, 1998.

[8] H. Tomiyamay and N. D. Dutt, "Program path analysis to bound cache-related preemption delay in preemptive real-time systems," *In CODES, 2000.*

[9] S. Altmeyer and C. Burguiere, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture (JSA), 2011, Elsevier.*

[10] H. S. Negi, T. Mitra, and A. Roychoudhury, "Accurate estimation of cache related preemption delay," *In CODES, 2003.*

[11] Y. Tan and V. Mooney, "Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems," *In SCOPES, 2004.*

[12] J. Staschulat and R. Ernst, "Scalable precision cache analysis for real-time software," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 4, September 2005.

[13] H. Ramaprasad and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," *In RTAS, 2006.*

[14] A. Burns, *Preemptive priority-based scheduling: an appropriate engineering approach*, 1995.

[15] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," *In ECRTS, 2005.*

[16] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," *In the International Conference on Real Time Computing Systems and Applications, 1999.*

[17] J. Simonson and J. Patel, "Use of preferred preemption points in cache based real-time systems," *In IPDPS, 1995*, pp. 316–325.

[18] R. Bril, S. Altmeyer, M. V. D. Heuvel, R. Davis, and M. Benham, "Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with preemption thresholds," *In RTSS, 2014.*

[19] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Preemption points placement for sporadic task sets," *In ECRTS, 2010.*

[20] J. M. Marinho, V. Nelis, S. M. Petters, and I. Puaut, "Preemption delay analysis for floating non-preemptive region scheduling," *In EDDA, 2012.*

[21] B. Peng, N. Fisher, and M. Bertogna, "Explicit preemption placement for real-time conditional code," *In ECRTS, 2014.*

[22] MRTC benchmarks. [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[23] Gaisler Bare-C Cross Compiler (BCC). [Online]. Available: http://gaisler.com/index.php/downloads/compilers

[24] GRSIM. [Online]. Available: http://gaisler.com/index.php/products/simulators/grsim

[25] AbsInt a^3 WCET Tool. [Online]. Available: http://www.absint.com/a3/index.htm

[26] Paper Programs and Data Repository. [Online]. Available: https://github.com/ctessler/superblocks/tree/master/study

[27] W. Lunniss, S. Altmeyer, C. Maiza, and R. Davis, "Integrating cache related pre-emption delay analysis into edf scheduling," *In RTAS, 2013.*

IX. Appendix

The appendix presents two supporting examples covered in the two subsections below. The first example demonstrates how LCBs are computed. The second example illustrates the interdependence of LCBs.

A. Example LCB Calculation

To illustrate our approach for computing LCBs, consider the following example. Assume we have two tasks, τ_1 and τ_2 with UCBs and ECBs for each listed in Figure 9. Please note that we have not shown the ECBs, UCBs, or AUCBs in the following figures for δ_i^0 as it is a dummy basic block with no elements for each of the aforementioned sets.

Task ID	Evicting Cache Blocks ECB(δ_i^j)				
	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5
τ_1	{1,2}	{3,4,8}	{4,5,6,8}	{1,2,7,8}	{1,2,7,8}
τ_2	{1,9}	{3,10}	{11,12}	{5,7,13}	{1,3,7,8}

Task ID	Useful Cache Blocks $UCB_{out}(\delta_i^j)$				
	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5
τ_1	{1,2}	{1,2,4,8}	{1,2,8}	{1,2,7,8}	{1,2,7,8}
τ_2	{1}	{1,3}	{1,3}	{1,3,7}	{1,3,7,8}

Fig. 9: Taskset ECBs and UCBs.

The computation for LCBs uses the accessed useful cache blocks (AUCBs) since the cache blocks that are re-loaded during execution of the non-preemptive region between preemption points is a function of the memory that is explicitly accessed by the preempted task. The computed AUCBs for each task is shown in Figure 10. In accordance with Equation 7 one can readily see that the AUCBs are simply the intersection of the UCBs and ECBs for each basic block.

Task ID	Accessed Useful Cache Blocks $AUCB_{out}(\delta_i^j)$				
	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5
τ_1	{1,2}	{4,8}	{8}	{1,2,7,8}	{1,2,7,8}
τ_2	{1}	{3}	{1,3}	{7}	{1,3,7,8}

Fig. 10: Taskset AUCBs.

In our example, assume preemptions are taken at basic blocks δ_1^2 and δ_1^4 for task τ_1. For simplicity, we calculate the LCBs associated with these two preemption points. For $LCB(\delta_1^2, \delta_1^4)$, we have $UCB(\delta_1^2) = \{1,2,4,8\}$. The second expression is the set of memory that is accessed in basic blocks δ_1^3 and δ_1^4, namely $\{8\} \cup \{1,2,7,8\} = \{1,2,7,8\}$ comprising the set of AUCBs. The third expression is the set of ECBs for task τ_2 where $ECB(\tau_2) = \{1,3,5,7,8,9,10,11,12,13\}$. Thus, $LCB(\delta_1^2, \delta_1^4)$ is given by the intersection of the three sets:

$$LCB(\delta_1^2, \delta_1^4) = \{1,2,4,8\} \cap \{1,2,7,8\} \cap \{1,3,5,7,8,9,10,11,12,13\} = \{1,8\}$$

The preemption cost $\gamma(\delta_1^2, \delta_1^4)$ for a BRT = 390μs is given by:

$$\gamma(\delta_1^2, \delta_1^4) = |\{1,8\}| \cdot 390 = 780\mu s$$

Using the same method, $LCB(\delta_1^4, \delta_1^5)$ is given by:

$$LCB(\delta_1^4, \delta_1^5) = \{1,2,7,8\} \cap \{1,2,7,8\} \cap \{1,3,5,7,8,9,10,11,12,13\} = \{1,7,8\}$$

The preemption cost $\gamma(\delta_1^4, \delta_1^5)$ for a BRT = 390μs is given by:

$$\gamma(\delta_1^4, \delta_1^5) = |\{1,7,8\}| \cdot 390 = 1170\mu s$$

Using the method illustrated here, the preemption cost matrix entries for each pair of basic blocks are computed in a similar fashion and used as input to our preemption point placement algorithm.

B. Example of LCB Interdependence

To further exemplify the interdependence of preemption points, consider the example shown below. In order to account for all re-loaded cache blocks (LCBs), preemptions are always included at the first basic block δ_i^0 and the last basic block $\delta_i^{N_i}$ as shown in Figure 11. This is commensurate with the preemptions that occur before and after the task executes. Assume we have two tasks where τ_2 contains four basic blocks which may be preempted by task τ_1. For simplification, assume that the ECBs of task τ_1 evicts all UCBs of task τ_2. Let us further assume that we have $\rho_2 = \{\delta_2^0, \delta_2^1, \delta_2^2, \delta_2^4\}$. Using our LCB computation approach, only the re-loaded lines as captured in the terms $LCB(\delta_2^1, \delta_2^2)$ and $LCB(\delta_2^2, \delta_2^4)$ are included in the C_2 computation. $LCB(\delta_2^0, \delta_2^1) = 0$ as no LCBs have been cached until after execution of basic block δ_2^1.

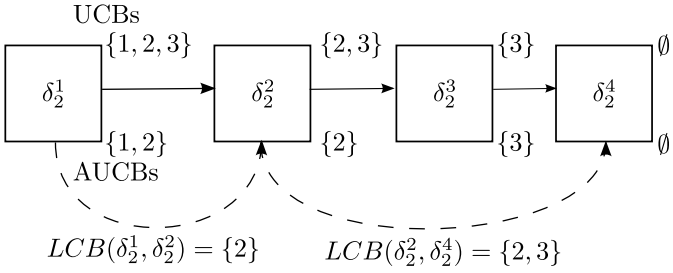

Fig. 11: LCB Interdependence