# Optimal Preemption Point Placement Using Enhanced Cache Related Preemption Delay

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 supporting preemption, the cache related preemption delay (CRPD) can represent a significant component (up to 44% as documented in research literature) [1] [2] [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 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 leveraging the enhanced CRPD computation to more accurately model task WCET and maximize schedulability as compared to existing preemption point placement approaches. We propose a revised 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 using the more accurate CRPD and associated task WCET values.**

## I. Introduction

Real-time systems differ from traditional computing systems in that one or more 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 prominent research on schedulability analysis. Paramount to schedulability analysis is accurate characterization of WCET. 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] [2] [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 $\tau_i$ is preempted by one or more higher priority tasks denoted $\tau_k$ whose execution results in the eviction of cache memory blocks that must be subsequently reloaded when task $\tau_i$ 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 quadratic time algorithm for selecting optimal preemption points for a linear basic block structure. A linear basic block structure implies that conditional logic and branches are fully contained within basic block boundaries. CRPD literature permits basic blocks connected by arbitrary control flow graph structures. The primary contributions outlined in this paper include improved accuracy for computing CRPD cost taking into account where preemptions actually occur, and a revised 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 and optimal preemption point placement 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 is 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 bench mark 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

### A. CRPD Calculation

Analyzing the preempted task, Lee et al. [5] [6] [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 the CRPD, which is simply the cardinality of the UCB set times the cache block reload time (BRT).

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. The set of UCBs and ECBs can be computed using control flow graph (CFG) analysis of reaching memory blocks (RMB) and live memory blocks (LMB) [5] [6] [7].

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. One of the limitations with the existing UCB based analysis methods is their representation of memory blocks that may reside in cache memory. This over-approximation was employed to realize a safe bounds on CRPD which is referred to as may cache [9].

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 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 the 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 fixed preemption point model and 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 [15] approach computes the maximum amount of blocking time denoted $Q_i$ for which a task $\tau_i$ 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 $\Pi_i$. A task will be preempted only if the preempting task has a nominal priority $p_k$ greater than the preemption threshold $\Pi_i$. 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 derivative 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. 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 useful cache block 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.

In contrast, Bertogna et. al. [18] achieved an optimal preemption point placement algorithm with quadratic 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.

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. 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 $C_i^{NP} > Q_i$.

## III. System Model

Our system contains a task set $\tau$ of $n$ periodic or sporadic tasks ($\tau_1$, $\tau_2$, ..., $\tau_j$, ..., $\tau_n$) each scheduled on a single processor. Each task $\tau_i$ is characterized by a tuple ($\phi_i$, $C_i^{NP}$, $D_i$, $T_i$, $J_i$, $P_i$) where $\phi_i$ is the starting time (also known as the phase), $C_i^{NP}$ is the non-preemptive worst case execution time, $D_i$ is the relative deadline, $T_i$ is the inter-arrival time or period, $J_i$ is the release jitter and $P_i$ is the uniquely assigned static fixed priority. Each task $\tau_i$ creates an infinite number of jobs, with the first job arriving at starting time $\phi_i$ 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 ($\delta_i^1$, $\delta_i^2$, ..., $\delta_i^{N_i}$) with the total number of basic blocks given by N = $\Sigma_i N_i$. In our linear 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 control flow graph (CFG) connected in a linear sequence by edges $E$ representing the execution sequence of one or more job instructions. Preemptions are permitted at basic block boundaries. We introduce the basic block notation $\delta_i^j$ 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 task release jitter $J_i$ is defined as the maximum time between the task arriving for execution and it being released to a state of being ready to execute. In our work, we assume the release jitter $J_i$ = 0. The processor utilization $U_i$ of task $\tau_i$ is given by

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

Blocking time $\beta_i$ is the time which task $\tau_i$ is subject to accounting for the maximum time that a lower priority task either executes non-preemptively or holds a resource that is shared with task $\tau_i$ or any other task of equal or higher priority. A direct-mapped cache is assumed, though the techniques described here can be readily extended to set-associative caches. Tasks may be preempted by multiple higher-priority tasks identified by the variable $k$ where

$$k \in hp(i) = \{k | P_k > P_i\}. \tag{3}$$

comprising the set of higher priority tasks for task $\tau_i$. This set needs to be filtered by tasks that may preempt during the execution window of task $\tau_i$. The number of times task $\tau_k$ can preempt task $\tau_i$ during task $\tau_i$'s execution is given by the term

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

where $k$ is the index of the preempting task $\tau_k$ and $i$ is the index of the preempted task $\tau_i$. In a limited preemption approach, each task is permitted to execute non-preemptively for a maximum amount of time denoted by $Q_i$. Figure 1 shown below illustrates the linear basic block connection structure.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_i^0$ | $\xi_i^0$ | $b_i^1$ | $\xi_i^1$ | $b_i^2$ | $\xi_i^2$ | $b_i^3$ | $\xi_i^3$ | $b_i^4$ | $\xi_i^4$ | $b_i^5$ | $\xi_i^5$ | $b_i^6$ | $\xi_i^6$ |
| $\delta_i^0$ | | $\delta_i^1$ | | $\delta_i^2$ | | $\delta_i^3$ | | $\delta_i^4$ | | $\delta_i^5$ | | $\delta_i^6$ | |

Fig. 1: System Model.

## 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 re-loaded cache blocks (LCBs) due to higher priority task preemption by using the set of potential/chosen preemption locations. Consistent with the literature the sets of various cache blocks CBs 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-set $s \in \mathit{UCB}(\tau_i)$ if and only if $\tau_i$ has a useful cache block in cache-set $s$. Note this definition of *UCB* embodies a task level view. Cache-set $s \in \mathit{UCB}_{out}(\delta_i^j)$ if and only if $\delta_i^j$ has a useful cache block in cache-set $s$ where

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

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

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

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

To capture the notion of what we are attempting to count, consider the set of replaced cache blocks (RCBs).

**Definition 3. Replaced cache block (RCB)**: A memory block of the preempted task is called a replaced cache block, if it is replaced during the execution of a preempting task.

$\mathit{RCB}_{out}(\delta_i^j, \tau_k)$ denotes preemption of task $\tau_i$ by task $\tau_k$ occurring immediately after basic block $\delta_i^j$, where task $\tau_i$ at basic block location $\delta_i^j$ is a potential/selected preemption point in our preemption point placement algorithm described later. We compute the set of replaced cache blocks (RCBs) as follows:

$$\mathit{RCB}_{out}(\delta_i^j, \tau_k) = \mathit{UCB}_{out}(\delta_i^j) \cap \mathit{ECB}(\tau_k). \tag{7}$$

If a cache block is evicted the cost is paid where the preemptions occur. In order to determine which cache blocks are reloaded once preemption occurs, we introduce the notion of an accessed useful cache block (AUCB).

**Definition 4. 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 $\delta_i^j$ for task $\tau_i$.

The term $\mathit{AUCB}_{out}(\delta_i^j)$ represents the useful cache blocks (UCBs) accessed by task $\tau_i$ at during execution of basic block at location $\delta_i^j$. 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:

$$\mathit{AUCB}_{out}(\delta_i^j) = \mathit{UCB}_{out}(\delta_i^j) \cap \mathit{ECB}(\delta_i^j). \tag{8}$$

where $\mathit{UCB}_{out}(\delta_i^j)$ is the set of useful cache blocks for task $\tau_i$ post basic block $\delta_i^j$ execution; and $\mathit{ECB}(\delta_i^j)$ denotes the set of cache blocks accessed in task $\tau_i$ during execution of basic block $\delta_i^j$. 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 5. 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 execution following a preempting task.

$\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}, \tau_k)$ denotes set set of cache blocks re-loaded during execution of the non-preemptive region between basic block $\delta_i^{curr}$ and basic block $\delta_i^{next}$, resulting from preemption of task $\tau_i$ by task $\tau_k$, where basic block location $\delta_i^{curr}$ and $\delta_i^{next}$ are the potential/selected preemption point and next potential/selected preemption point respectively in task $\tau_i$.

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, and the preempting task. As previously stated, there are two options in terms of where the cache block reload cost can be paid: 1) the cost can be paid where the next memory accesses occur; or 2) The cost can also be paid where the preemptions occur. In this paper, we use the cost paid where the preemptions occur as per below.

$$\begin{aligned}&\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}, \tau_k) = \\ &\left[\mathit{UCB}_{out}(\delta_i^{curr}) \ \cap \left[\cup_\lambda \mathit{AUCB}_{out}(\delta_i^\lambda)\right]\right] \cap \mathit{ECB}(\tau_k)\end{aligned} \tag{9}$$

where

$$\lambda \overset{\text{def}}{=} \{\nu | \nu \in [\delta_i^{curr}, \delta_i^{next}]\}$$

$\mathit{AUCB}_{out}(\delta_i^x)$ represents the accessed useful cache blocks or memory of interest accessed by the preempted task $\tau_i$ at basic block $\delta_i^x$; $\delta_i^{curr}$ represents the current selected preemption point where $\delta_i^{curr} \in \rho(\tau_i)$ and $\delta_i^{next}$ represents the current selected preemption point where $\delta_i^{next} \in \rho(\tau_i)$. This formula for $\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}, \tau_k)$ results in the accounting of loaded cache blocks where the preemption occurs. Note that this notation assumes a linear 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 per below.

$$\gamma(\delta_i^{curr}, \delta_i^{next}, \tau_k) = |\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}, \tau_k)| \cdot BRT. \quad (10)$$

where BRT is the cache block reload time; $\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}, \tau_k)$ represents the loaded cache blocks or memory accessed by the preempted task $\tau_i$ at basic block $\delta_i^{curr}$ caused by preempting task $\tau_k$.

## V. Integrated WCET/CRPD Calculation

The modified preemption cost as a function of the current and next preemption points and the preempting task $\tau_k$ is given by:

$$\begin{aligned}\xi_i(\delta_i^{curr}, \delta_i^{next}, \tau_k) \; &= \gamma_i(\delta_i^{curr}, \delta_i^{next}, \tau_k) + \pi + \sigma + \\ &\eta(\gamma_i(\delta_i^{curr}, \delta_i^{next}, \tau_k)).\end{aligned} \quad (11)$$

where $\pi$ is the pipeline cost, $\sigma$ is the scheduler processing cost, and $\eta()$ is the front side bus contention resulting from the cache reload interference as described in [1] [2] [3]. Commensurate with our worst case analysis, the preemption overhead cost as a function of the current and next selected preemption points is:

$$\xi_i(\delta_i^{curr}, \delta_i^{next}) = \mathit{max}_{k \in hp(i)}[\xi_i(\delta_i^{curr}, \delta_i^{next}, \tau_k)]. \quad (12)$$

The output of our algorithm is an optimal set of preemption points 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 given by:

$$C_i = B_i^{0,N_i}(\rho_i) = C_i^{NP} + \sum_{m=1}^{|\rho_i|-1} [\xi_i(\rho_i^m, \rho_i^{m+1})]. \quad (13)$$

where $\rho_i$ is the set of selected preemption points for task $\tau_i$:

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

and $B_i^{k,m}$ is the WCET with preemption overhead of basic blocks $k$ to $m$ of task $\tau_i$ as given by:

$$\begin{aligned}B_i^{k,m}(\rho_i) = \min_{\delta_i^j \in \tau_i} \Big[ & B_i^{j-1}(\rho_i(\delta_i^{j-1})) + \\ & \xi_i(\delta_i^j, \rho_i^{next}(\delta_i^j)) + \sum_{n=k}^{m} b_i^n \Big]\end{aligned} \quad (14)$$

where the term $\rho_i^{next}(\delta_i^j)$ is the next potential/selected preemption point from basic block $\delta_i^j$:

$$\begin{aligned}\rho_i^{next}(\delta_i^k) \overset{\text{def}}{=} \{ & \delta_i^j | \delta_i^k = \rho_i^m \wedge \delta_i^j = \rho_i^{m+1} \\ & \text{for some } m \in [1, N_i - 1]\}\end{aligned} \quad (15)$$

The maximum blocking time $\beta_i$ 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] [18]. The circular dependency between the maximum blocking time $\beta_i$ 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 1.

**Algorithm 1** Iterative Schedulability and Preemption Point Placement 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 Baruah algorithm to obtain the maximum
           non-preemptive region $Q_i$ for each task.
5:     Select optimal preemption points using our
           dynamic programming algorithm.
6:     Compute the CRPD and the preemptive WCET
           $C_i$ from the selected preemption points.
7: until the selected preemption points do not change.
```

## VI. Preemption Point Placement Algorithm

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 $\tau_i$. The objective is to select a subset of preemption points that minimizes each tasks WCET + CRPD parameter $C_i$. The selection of optimal preemption points is subject to the constraint that no non-preemptive region in task $\tau_i$ exceeds the maximum allowable non-preemption region parameter $Q_i$:

$$\Psi_i(\rho_i) = \left\{ \begin{array}{ll} \text{True}, & \text{if } q_i^m(\rho_i) \le Q_i \text{ for } m \in [1, |\rho_i| - 1] \\ \text{False}, & \text{otherwise} \end{array} \right\} \quad (16)$$

where $q_i^m(\rho_i)$ represents the $m^{th}$ non-preemptive-region (NPR) time for task $\tau_i$:

$$q_i^m(\rho_i) = \left[ \xi_i(\rho_i^m, \rho_i^{m+1}) + \sum_{n=\rho_i^m}^{\rho_i^{m+1}} b_i^n \right] \quad (17)$$

In accordance with the recursive nature of equation (14) we propose an O(N!) recursive algorithm shown in Algorithm 2 for computing the optimal preemption points. While the recursive formulation is clearly inefficient, it is helpful in developing an understanding of how the algorithm works. Starting with the first basic block and for each successive basic block $\delta_i^m, m \in [1, N_i]$, the overall WCET cost is computed for two cases: 1) $\delta_i^m \in \rho_i$, and 2) $\delta_i^m \notin \rho_i$. At each step of the algorithm the set $\rho_i$ must conform to the constraint of equation (16). Once basic block $\delta_i^{N_i}$ has been examined, the set of selected preemption points is given by $\rho_i = \rho_i^{(N_i)}$. The WCET cost with basic block $\delta_i^m$ included in the set of potential preemption points is given by:

$$p_{COST}(\delta_i^m) \;=\; B_i^{0,m}(\rho_i) + B_i^{m,N_i}(\rho_i) \qquad (18)$$

The WCET cost with basic block $\delta_i^m$ excluded from the set of potential preemption points is given by:

$$n_{COST}(\delta_i^m) \;=\; B_i^{0,m-1}(\rho_i) + b_i^m + B_i^{m+1,N_i}(\rho_i) \qquad (19)$$

**Algorithm 2** Recursive Optimal Preemption Point Placement

```
Step 0:
$\rho_i^{(0)} \leftarrow \{\delta_i^0, \delta_i^{N_i}\}$;
if $\Psi_i^{N_i}(\{\delta_i^0, \delta_i^1, \delta_i^2, ..., \delta_i^{N_i}\}) = False$ then
    return INFEASIBLE;
end if

Steps m = 1, ..., $N_i$:
$\rho_i^{(m)} \leftarrow \left\{ \begin{array}{c} \rho_i^{(m-1)}, if\ n_{COST}(\delta_i^m) < p_{COST}(\delta_i^m)\ \& \\ \Psi_i^m(\rho_i^{(m-1)}) = True \\ \rho_i^{(m-1)} \cup \delta_i^m,\ otherwise \end{array} \right\}$
```

The recursive formulation gives a preliminary algorithm description, however, it is computationally intractable. We now propose an $O(cN^2)$ dynamic programming algorithm for computing the optimal preemption points. In support of our dynamic programming algorithm, we propose the following theorem claiming optimal substructure of our solution formulation in terms of the WCET + CRPD cost $B_i$.

**Theorem 1.** *The WCET + CRPD cost variable $B_i$ utilized in our solution exhibits optimal substructure.*

We now prove the optimal substructure property of the WCET + CRPD cost $B_i$.

*Proof:* Let $B_i^{j,k}(\rho_i)$ with its corresponding selected preemption points denoted by $\rho_i^{j,k}$ be the optimal limited preemption execution cost solution from basic block $j$ to basic block $k$, and assume the optimal cost solution contains the basic block identified by $m$. Furthermore, let $B_i^{j,m}(\rho_i)$ with selected preemption points denoted by $\rho_i^{j,m}$ and $B_i^{m+1,k}(\rho_i)$ with selected preemption points denoted by $\rho_i^{m+1,k}$ be the optimal limited preemption execution cost from $j$ to $m$ and from $m+1$ to $k$, respectively, contained in the optimal solution to the original problem. Three additional constraints are imposed where $\rho_i^{j,m} \subseteq \rho_i^{j,k}$ and $\rho_i^{m+1,k} \subseteq \rho_i^{j,k}$ with $B_i^{j,m}(\rho_i) + B_i^{m+1,k}(\rho_i) = B_i^{j,k}(\rho_i)$. To prove optimal substructure, we need to prove that in order for the limited preemption execution cost $B_i^{j,k}(\rho_i)$ to be optimal, the limited preemption execution costs $B_i^{j,m}(\rho_i)$ and $B_i^{m+1,k}(\rho_i)$ must also be optimal solutions to their respective sub-problems.

Using proof by contradiction, assume there is a better solution $B_i^{'j,m}(\rho_i^{'})$ for the sub-problem of determining the optimal limited preemption execution costs from basic block j to basic block m, such that $B_i^{'j,m}(\rho_i^{'}) < B_i^{j,m}(\rho_i)$ and $\rho_i^{'j,m} \neq \rho_i^{j,m}$. Since the solution for selecting the optimal preemption points from basic block j to k will work regardless of which set of preemption points are used between basic block j and basic block k, we could then use the better solution $B_i^{'j,m}(\rho_i^{'})$ and $\rho_i^{'j,m}$ to arrive at a lower cost solution to the original problem. Thus we have the following:

$$B_i^{j,k}(\rho_i) \;=\; B_i^{j,m}(\rho_i) + B_i^{m+1,k}(\rho_i) \qquad (20)$$

$$B_i^{j,k}(\rho_i) \;>\; B_i^{'j,m}(\rho_i^{'}) + B_i^{'m+1,k}(\rho_i^{'}) \qquad (21)$$

However this contradicts the original definition that $B_i^{j,k}(\rho_i)$ and $\rho_i^{j,k}$ form an optimal solution to the problem. Therefore, the optimal limited preemption execution cost from basic block j to basic block k contained in the original solution $B_i^{j,k}(\rho_i)$ must be an optimal solution to the sub-problem of determining the optimal limited preemption execution cost solution from basic block j to basic block k. The same argument works for the sub-problem of determining the optimal limited preemption execution cost solution from basic block m+1 to basic block k. Thus, the problem exhibits optimal substructure. ■

The algorithm is summarized in Algorithm 3 shown below. For each task $\tau_i$, 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 $\xi_i$. The preemption cost matrix $\xi_i$ is organized for each basic block $\delta_i^j$ and contains the preemption cost for all successor basic blocks of the task's control flow graph. The minimum (preemptive or non-preemptive) cost between all basic blocks is computed and stored in a matrix denoted $B_i$. The $B_i$ matrix is initially seeded with the non-preemptive cost for basic block pairs that satisfy the constraint $q_i(\delta_i^j, \delta_i^k) < Q_i$. All other entries are set to infinity. As we consider whether each basic block $\delta_i^k$ is in the set of optimal preemption points, the location of the previous basic block $\delta_i^j$ with minimal preemption cost is stored in an array denoted $\rho_{prev}$. The algorithm examines each basic block from $\delta_i^1$ to $\delta_i^{N_i}$ to minimize the preemption cost by traversing backwards from the current basic block $\delta_i^k$ under consideration in order to find the basic block $\delta_i^j$ with minimal preemption cost subject to the constraint $q_i(\delta_i^j, \delta_i^k) < Q_i$. 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 $\delta_i^m$. Basic blocks $\delta_i^{N_i}$ and $\delta_i^m$ are added to the optimal preemption point set $\rho_i$. This basic block hopping process continues until basic block $\delta_i^0$ is reached. The set $\rho_i$ contains the complete list of selected optimal preemption points. To exemplify how our algorithm works,

**Algorithm 3** D.P. Optimal Preemption Point Placement

```
function $Select\_Optimal\_PPP(N_i,b_i,Q_i,\xi_i)$
    $C_i^{NP} \leftarrow \infty\ q_i \leftarrow \infty\ B_i \leftarrow \infty\ \rho_{prev} \leftarrow \{\delta_i^0\}$;
    if $b_i^k > Q_i$ for some $k \in [1, N_i]$ then
        return INFEASIBLE;
    end if
    for $k : 2 \le k \le N_i$ do
        $Compute\_PPCost(\delta_i^{k-1},\delta_i^k)$;
        for $j : k-1 \ge j \ge 1$ and $q_i(\delta_i^j, \delta_i^k) < Q_i$ do
            $Compute\_PPCost(\delta_i^0,\delta_i^j)$;
            $Compute\_PPCost(\delta_i^j,\delta_i^k)$;
            $P_{cost} \leftarrow B_i(\delta_i^0, \delta_i^j) + B_i(\delta_i^j, \delta_i^k)$;
            if $P_{cost} < B_i(\delta_i^0, \delta_i^k)$ then
                $B_i(\delta_i^0, \delta_i^k) \leftarrow P_{cost}$;
                $\rho_{prev}(\delta_i^k) \leftarrow \delta_i^j$;
            end if
        end for
    end for
    $\rho_i \leftarrow Compute\_PPSet(N_i, \rho_{prev})$;
    return FEASIBLE;
end function

function $Compute\_PPCost(\delta_i^j,\delta_i^k)$
    if $q_i(\delta_i^j, \delta_i^k) = \infty$ then
        Compute $\xi_i(\delta_i^j,\delta_i^k)$ using Equation (12);
        Compute $C_i^{NP}(\delta_i^j,\delta_i^k)$;
        Compute $q_i(\delta_i^j,\delta_i^k)$;
        if $q_i(\delta_i^j,\delta_i^k) \le Q_i$ then
            $B_i(\delta_i^j, \delta_i^k) \leftarrow q_i(\delta_i^j,\delta_i^k)$;
            $\rho_{prev}(\delta_i^k) \leftarrow \delta_i^j$;
        else
            if $j - k = 1$ then
                return INFEASIBLE;
            end if
        end if
    end if
end function

function $Compute\_PPSet(N_i,\rho_{prev})$
    Computes $\rho_i$ from $\rho_{prev}$ (Details omitted)
end function
```

consider the following example. Let $N_i = 6$ and $Q_i = 12$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | - | 3 | - | 2 | - | 2 | - | 2 | - | 2 | - | 3 | - |
| $\delta_i^0$ | $\xi_i^0$ | $\delta_i^1$ | $\xi_i^1$ | $\delta_i^2$ | $\xi_i^2$ | $\delta_i^3$ | $\xi_i^3$ | $\delta_i^4$ | $\xi_i^4$ | $\delta_i^5$ | $\xi_i^5$ | $\delta_i^6$ | $\xi_i^6$ |

| $\xi_i$ | $\delta_i^0$ | $\delta_i^1$ | $\delta_i^2$ | $\delta_i^3$ | $\delta_i^4$ | $\delta_i^5$ | $\delta_i^6$ |
|---|---|---|---|---|---|---|---|
| $\delta_i^0$ | **0** | 1 | 2 | 4 | 4 | 3 | 2 |
| $\delta_i^1$ | - | **1** | 3 | 5 | 5 | 4 | 3 |
| $\delta_i^2$ | - | - | **2** | 6 | 6 | 5 | 4 |
| $\delta_i^3$ | - | - | - | **4** | 6 | 7 | 6 |
| $\delta_i^4$ | - | - | - | - | **4** | 5 | 6 |
| $\delta_i^5$ | - | - | - | - | - | **3** | 5 |
| $\delta_i^6$ | - | - | - | - | - | - | **2** |

Fig. 2: Algorithm Example.

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 $C_i^{NP}(\delta_i^j,\delta_i^k)$. Using this information, the combined WCET and CRPD costs for each basic block pair is computed and stored in a matrix denoted $q_i$. 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 pair and stored in a matrix denoted $B_i$. The $B_i$ matrix stores the non-preemptive execution cost above the diagonal and the preemptive execution cost below the diagonal. 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 pair, the predecessor preemption point is updated in the $\rho_{prev}$ array which keeps track of the selected predecessor preemption point for each basic block forming a daisy chain containing the entire set of selected preemption points. The final results are illustrated in Figures 3 and 4 below.

| $q_i$ | $\delta_i^0$ | $\delta_i^1$ | $\delta_i^2$ | $\delta_i^3$ | $\delta_i^4$ | $\delta_i^5$ | $\delta_i^6$ |
|---|---|---|---|---|---|---|---|
| $\delta_i^0$ | **0** | 4 | 7 | 11 | 13 | 14 | 16 |
| $\delta_i^1$ | - | **4** | 8 | 12 | 14 | 15 | 17 |
| $\delta_i^2$ | - | - | **4** | 10 | 12 | 13 | 15 |
| $\delta_i^3$ | - | - | - | **6** | 10 | 13 | 15 |
| $\delta_i^4$ | - | - | - | - | **6** | 9 | 13 |
| $\delta_i^5$ | - | - | - | - | - | **5** | 10 |
| $\delta_i^6$ | - | - | - | - | - | - | **5** |

Fig. 3: Combined WCET and CPRD Costs.

| $B_i$ | $\delta_i^0$ | $\delta_i^1$ | $\delta_i^2$ | $\delta_i^3$ | $\delta_i^4$ | $\delta_i^5$ | $\delta_i^6$ |
|---|---|---|---|---|---|---|---|
| $\delta_i^0$ | - | 3 | 5 | 7 | 9 | 28 | 38 |
| $\delta_i^1$ | 4 | - | 5 | 7 | 9 | 29 | 39 |
| $\delta_i^2$ | 7 | 8 | - | 4 | 6 | 8 | 31 |
| $\delta_i^3$ | 11 | 12 | 10 | - | 4 | 6 | 20 |
| $\delta_i^4$ | 19 | 20 | 12 | 10 | - | 4 | 7 |
| $\delta_i^5$ | 28 | 29 | 21 | 19 | 9 | - | 5 |
| $\delta_i^6$ | 38 | 39 | 31 | 29 | 19 | 10 | - |

| $\rho_{PREV}$ | $\delta_i^0$ | $\delta_i^1$ | $\delta_i^2$ | $\delta_i^3$ | $\delta_i^4$ | $\delta_i^5$ | $\delta_i^6$ |
|---|---|---|---|---|---|---|---|
| | $\delta_i^0$ | $\delta_i^0$ | $\delta_i^0$ | $\delta_i^0$ | $\delta_i^2$ | $\delta_i^4$ | $\delta_i^5$ |

| $\rho_i$ | $\delta_i^0$ | $\delta_i^1$ | $\delta_i^2$ | $\delta_i^3$ | $\delta_i^4$ | $\delta_i^5$ | $\delta_i^6$ |
|---|---|---|---|---|---|---|---|
| | $\delta_i^0$ | | $\delta_i^2$ | | $\delta_i^4$ | $\delta_i^5$ | $\delta_i^6$ |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 3 | 0 | 2 | 6 | 2 | 0 | 2 | 5 | 2 | 5 | 3 | 2 |
| $\delta_i^0$ | $\xi_i^0$ | $\delta_i^1$ | $\xi_i^1$ | $\delta_i^2$ | $\xi_i^2$ | $\delta_i^3$ | $\xi_i^3$ | $\delta_i^4$ | $\xi_i^4$ | $\delta_i^5$ | $\xi_i^5$ | $\delta_i^6$ | $\xi_i^6$ |

Fig. 4: Algorithm Results.

## 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 selected from Malardalen University of Sweden's WCET benchmark suite [19]. Each task was built using Gaisler's Bare-C Cross Compiler [20] for the GRSIM LEON3 [21] simulated target.

Within a task, variance in the number of shared UCBs between program points is related to the execution flow of the task through each of the points. For instance, one path to $\varrho_i^j$ may include only $\varrho_i^h$, while another path to $\varrho_i^j$ may contain $\varrho_i^h$ followed by $\varrho_i^m$. The set of UCBs shared between $\varrho_i^h$ and $\varrho_i^j$ may differ based on the path between them. If the paths differ one of them may result in a greater number of shared UCBs between the points. To accurately upper-bound the shared UCBs between points requires complete control flow information.

Tasks were first analyzed using AbsInt's a$^3$ WCET [22] to determine the set of basic blocks. Next, the basic blocks $\{\delta_i^1, \delta_i^2, ..., \delta_i^{N_i}\}$ were serialized by recording their order during execution. Program points $\{\varrho_i^1, \varrho_i^2, ..., \varrho_i^{N_i}\}$ were assigned by setting $\varrho_i^j$ to the address of the final instruction of each basic block $\delta_i^j$ for $j \in [0, N_i]$.

Each program point $\varrho_i^j$ served as a breakpoint when running the task on the simulator. As the task was executed, at each breakpoint $\varrho_i^j$, the instruction and data cache states were saved as $\Upsilon_I(\delta_i^j)$ and $\Upsilon_D(\delta_i^j)$. Given the limitations of the simulator and a$^3$ it was not possible record the actual control flow. Thus, calculating an accurate over-estimate of the UCBs shared between program points was not possible. A selection of cache snapshots were chosen to representation the actual behavior.

The representative cache contents were selected as the final visit of each program point. A program point $\varrho_i^j$ may be visited multiple times during a tasks execution. The UCBs shared between $\varrho_i^j$ and a later point $\varrho_i^k$ could change with each visit of $\varrho_i^j$. During the final visit of $\varrho_i^j$ the instruction and data cache contents were captured, and recorded as $\Upsilon_I(\delta_i^j)$ and $\Upsilon_D(\delta_i^j)$. From these representative snapshots, a representative over-estimate of shared UCBs were made.

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

$$UCB(\varrho_i^j, \varrho_i^k) = \bigcap_{m=j}^{k-1} \Upsilon_D(\delta_i^m)$$

*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 [23] thereby permitting the research community to reproduce and leverage our work as needed.

*2) Results:* The results are presented as a comparison between the our proposed method and the Bertogna approach. For a program point $\varrho_i^j$ the Bertogna [4] approach defines the UCBs (and therefor the CRPD) as:

$$\max\{UCB(\varrho_i^j, \varrho_i^k) | j < k\}$$

To determine the maximum benefit of the new approach, the best case scenario is considered. When the preemption point is selected with the fewest number of UCBs based upon previous preemption. For $\varrho_i^j$ the determination is made by:

$$\min\{UCB(\varrho_i^j, \varrho_i^k) | j < k\}$$

In the following graphs, each point in the graph represents two points in the program. The first point of the program $\varrho_i^j$ is fixed by the x-axis. The y-axis indicates the shared UCB count with a later program point. The first graph, which represents the recursion benchmark's data cache, is for illustration. At program point four, the Bertogna approach finds 24 shared UCBs between point four and point five. The proposed method selects program point seven with a shared UCB count of 14. To compare the two approaches their

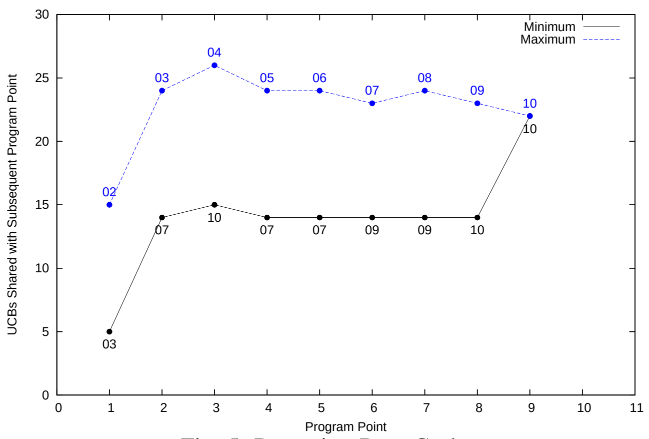

Fig. 5: Recursion Data Cache.

selection of two program points is simulated by assuming $Q_i$ is larger than the task's total execution time. Bertogna's approach would select the lowest value of the dashed line. The proposed approach would select the lowest value of the solid line. The difference between these two UCB counts is the benefit provided by the proposed approach.

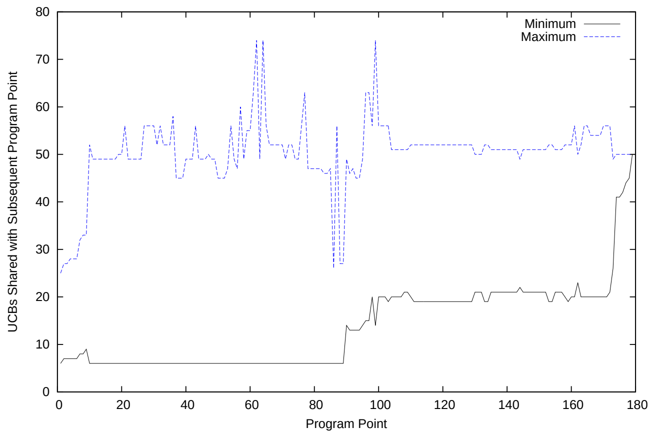

Fig. 6: LMS Data Cache.

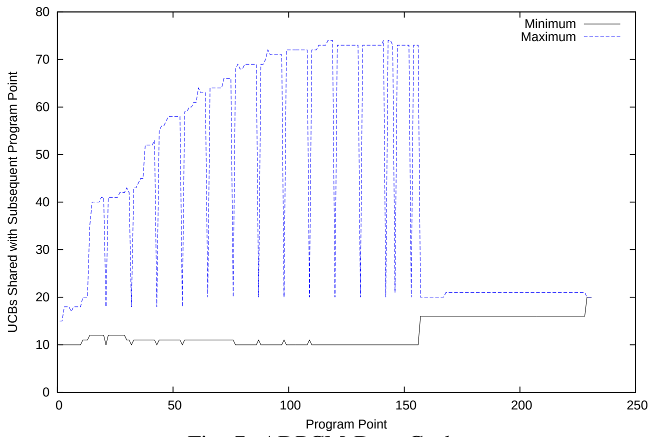

Fig. 7: ADPCM Data Cache.

Examining the graphs for all tasks and caches (instruction and data) there are a few common traits. The minimum shared UCBs sharply increases at the end of the tasks execution. This is due to the nature of the over-estimation of shared UCBs and the tasks, the number of instructions in the basic block between the final program points is relatively small; limiting the number of cache lines that could be filtered by the intersection.

This behavior limits the appropriate type of analysis. The preceding table shows the maximum benefit of the proposed approach. If the minimum benefit were considered it would always be zero; since the difference in estimated shared UCBs converges at the final basic block.

Drastic spikes downwards in the shared UCB counts for the minimum and maximum approaches coincide with function call boundaries, or large conditional blocks. At these boundaries, the maximum and minimum approaches have similar UCB counts.

There is a sharp upward spike in the early program points for the maximum approach. This is likely due to the early initialization blocks built into tasks. The minimum approach shows a clear benefit of selecting a preemption point outside of the early initialization section.

## B. *Breakdown Utilization*

The previous analysis illustrates the benefit of the proposed approach for an individual task, which does not illustrate the schedulability benefit. To determine the schedulability benefit a second case study was performed. Focusing on the breakdown utilization when comparing the UCB only approach for EDF [24]. For convenience the UCB Only approach for EDF will be referred to as $UOE$, and Explicit Preemption Point Placement 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 a task $j$, the utilization of the task $j$ including the CRPD of the task, and the utilization of the task set respectively as documented in Lunniss *et al.* [24]. A task set is schedulable when $U^* \leq 1$.

The task set from which the breakdown utilization benefit is calculated comes, again, from the MRTC suite [19]. Borrowing the technique from [24], each task has its deadline (and therefor period) set to $T_i = u \cdot C_i$ where $u$ is a constant. The constant, $u$, begins at the number of tasks (ten) and is increased in steps of .25 until the task set becomes schedulable. Incremental negative adjustments are then made to determine when the set becomes unschedulable, indicating the breakdown utilization. For each task, the set of shared UCBs are calculated at each program point. Taking the maximum shared UCB count as $UCB_k$ for any task is safe and appropriate for calculating $U^*$. Similarly, for $EPP$, the shared UCB counts obtained in the earlier evaluation serve as input for the Explicit Preemption Point 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 $BRT$ is set to 7.8 $\mu s$, the refresh time of main memory. The breakdown utilization determination leverages our iterative schedulability and preemption point placement algorithm as outlined in the following steps below as given in Algorithm 4. Using ten

**Algorithm 4** 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 1
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.
```

tasks, the breakdown utilization comparison between $UOE$, $BertogaEPP$, and $EPP$ are summarized in the figure below. Compared to [24] this evaluation of $UOE$ has a breakdown utilization 17% lower. This is likely due to the

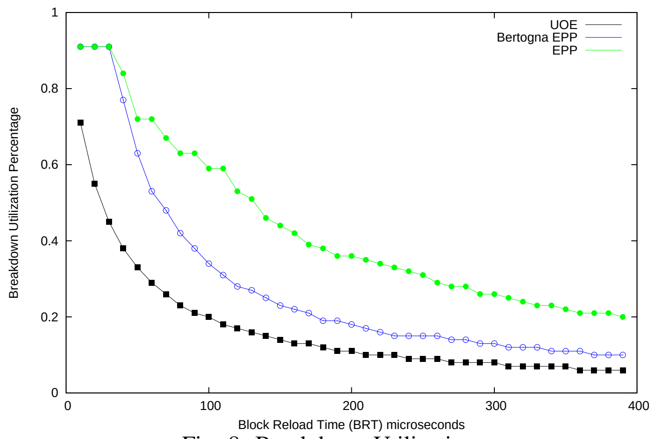

Fig. 8: Breakdown Utilization.

(differing) selection of tasks and estimation of UCBs. The data used in [24] does not include UCB information per basic block, which is required for $EPP$. Necessitating the generation of new UCB estimations per task.

## VIII. Conclusion

In this paper we presented an enhanced approach for calculating the CRPD taking into account the selected preemption points resulting in greater accuracy. Using the enhanced 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 method. 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 Malardalen University of Sweden's WCET benchmark suite [19]. 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. In future work, we plan to 1) a schedulability comparison of synthetic task set for various preemption models, and 2) remove the linear basic block restriction thereby permitting arbitrarily connected basic block structures.

## References

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

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

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

[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 Proceedings RTSS, 1996, IEEE.*

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

[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 Proceedings CODES, 2000, ACM.*

[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 Proceedings CODES, 2003, ACM.*

[11] Y. Tan and V. Mooney, "Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems," *In Proceedings 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 Proceedings RTAS, 2006, IEEE.*

[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 Proceedings ECRTS, 2005, IEEE.*

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

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

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

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

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

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

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

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

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