Realizing Improved Preemption Placement in Real-Time Program Code with Interdependent Cache Related Preemption Delay

John Cavicchio and Nathan Fisher
Department of Computer Science, Wayne State University
{ba6444, fishern}@wayne.edu

***Abstract*— The benefits of the limited preemption scheduling model serve to minimize preemption overhead while enabling cooperative scheduling between real-time tasks. Preemption point placement (PPP) algorithms are employed to select a suitable subset of preemption locations that optimize task worst case execution time (WCET). Combining PPP with schedulability analysis algorithms extends limited preemption scheduling to real-time task sets. Existing PPP algorithms are pessimistic due to a simplifying assumption made in calculating cache related preemption delay (CRPD): the cost of each preemption is assumed to be independent of the location of selected adjacent preemption points. In this work, we remove this simplifying assumption and develop an improved PPP algorithm based upon our interdependent CRPD calculation that can be applied to common application code structures (e.g., conditional branches, loops, etc.). A case study using the MRTC benchmarks will demonstrate significantly improved task set schedulability via our proposed interdependent CRPD PPP algorithm.**

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

I. Introduction

The utility of real-time system computations depends on two important properties, correctness and timeliness. The timeliness property (the subject of schedulability analysis) is concerned with ensuring real-time task computations are completed within required deadlines. Designers of real-time systems must choose the scheduling paradigm that will ultimately determine if the real-time task set will meet its timeliness objectives. The available choices are 1) non-preemptive scheduling, 2) fully preemptive scheduling, and 3) limited preemption scheduling. Non-preemptive scheduling suffers from blocking of high priority tasks and fully preemptive scheduling suffers from substantial preemption overhead (up to 44% [1]–[3] of a tasks WCET) each approach degrading task set schedulability. Limited preemption scheduling attempts to 1) reduce blocking by limiting the number of allowed preemptions, maximizing non-preemptive task execution and 2) reduce preemption overhead via non-preemptive regions. Regardless of the chosen scheduling paradigm, effective schedulability analysis of real-time task sets mandates accurate WCET and CRPD estimates. In this paper, the recognized benefits of limited preemption scheduling motivate our work on PPP algorithms.

The importance of CRPD in schedulability analysis stems from it comprising the majority of preemption overhead. CRPD occurs when a task denoted τ_i is preempted by one or more higher priority tasks denoted τ_k. The execution of high priority tasks results in the eviction of cache memory blocks that must be subsequently reloaded when task τ_i resumes execution. Two primary models of CRPD computation exist, 1) the independent CRPD cost model, and 2) the interdependent CRPD cost model. The vast majority of CRPD research falls under the independent CRPD model. Here, costs are solely a function of the preemption location under consideration. Since the next preemption may occur at any forward point in the task code, independent CRPD methods must conservatively utilize the next code location corresponding to the maximum CRPD cost. The interdependent CRPD cost model, however, overcomes this limitation by considering and computing costs between each pair of task code locations thereby achieving more accuracy. A key factor in preemption location decisions, CRPD cost accuracy is of paramount importance to PPP algorithms.

PPP algorithms select preemption points for each task to 1) minimize the overall task WCET, and 2) ensure the execution time between adjacent preemptions is limited by the maximum non-preemptive region execution time. The maximum non-preemptive region execution time, denoted Q_i, is determined via task set schedulability analysis. The motivation behind our work is the utility of existing PPP algorithms are limited either by the less accurate CRPD costs or by assuming a linear code structure (i.e., no branches or loops are permitted) [4]–[7]. Our approach removes this linear code assumption and combines the interdependent CRPD cost model with an improved PPP algorithm thereby reducing overall task WCET. The benefits of our approach will be illustrated in a case study employing real-time tasks from the MRTC benchmark suite.

The rest of this paper is organized as follows. First, current limited preemption scheduling research efforts and related work is discussed in Section II. Section III describes our real-time task model terminology. The real-time conditional flow graph model is detailed in Section IV. Section V presents a formal problem statement describing the objective function for selecting preemptions in conditional flowgraphs. Section VI discusses our enhanced pseudo quartic time interdependent CRPD preemption point placement algorithm. A case study using MRTC benchmarks demonstrates improved task set schedulability in Section VII. Finally, Section VIII offers relevant conclusions along with proposed future work.

II. Related Work

Our proposed conditional PPP algorithm leverages elements from two prominent areas of real-time theory, CRPD calculation, and limited preemption scheduling. We now briefly summarize the prior work in each of these areas and describe how it relates to our proposed algorithm.

A. CRPD Calculation

The concept and algorithm for computing the set of useful cache blocks (UCBs) was proposed by Lee et al. [8] analyzing the preempted task. Similarly, the set of evicting cache blocks (ECBs) was realized by Tomiyamay and Dutt [9] analyzing the preempting task. Altmeyer and Burguiere [10] refined and presented more formal definitions of UCBs and ECBs. By convention, the preempting task's memory accesses (ECBs) will evict the preempted task's UCBs thereby resulting in non-negligible CRPD.

Tighter bounds on the CRPD computation via the intersection of the ECB and UCB sets were achieved by Negi et al. [11] and Tan and Mooney [12]. Staschulat and Ernst [13] employed a cache state reduction technique trading off CRPD accuracy or tightness for a reduction in computational complexity. CRPD analysis using memory access patterns was proposed by Ramaprasad and Mueller [14].

Altmeyer and Burguiere [10] addressed the over-approximation of cache misses in WCET analysis tools by introducing definitely-cached useful cache block (DC-UCB) [10]. DC-UCBs are cache blocks that must reside in cache memory.

Altmeyer et al. [15] introduced the ECB Union approach and a combined UCB Union and ECB Union approach shown to dominate earlier CRPD methods. The UCB Union and ECB Union multiset approaches were subsequently introduced to reduce the pessimism in CRPD analysis [16].

The above approaches assume that the CRPD cost of a preemption is computed independently to obtain a safe, conservative bound. Conversely, Cavicchio et al. [7] proposed an interdependent CRPD model using loaded cache blocks (LCBs) to improve the CRPD accuracy in PPP algorithms leading to tangible schedulability gains. Our paper leverages the interdependent CRPD model and extends preemption placement to conditional real-time code structures.

B. Limited Preemption Scheduling

Research in limited preemption scheduling attempts to address well known limitations of the non-preemptive and fully preemptive scheduling paradigms. Two limited preemption scheduling models are the deferred preemption scheduling model and the preemption threshold scheduling model.

First, the fixed preemption point model [17] and the floating point preemption model [18] are two distinct subcategories of deferred preemption scheduling. In the floating preemption point model [18], the currently executing task continues executing for a minimum of Q_i time units or runs to completion if the remaining execution time is less than Q_i. The parameter Q_i is computed via task set schedulability analysis [18] representing the maximum amount of blocking time that task τ_i may impose on higher priority tasks. Since the start of the non-preemptive execution region coincides with the arrival of a higher priority task, region locations are nondeterministic or floating. In contrast, the fixed point preemption model [17] differs from the floating preemption point model in that the task code preemptions are confined to pre-defined fixed locations and computed using an offline PPP algorithm.

Second, preemption threshold scheduling [19] employs a modified priority based scheme to determine when tasks may preempted. Each task in the preemption threshold scheduling [19] approach is assigned two distinct priority values, each for a different purpose. These priorities are known as the nominal static priority p_i and a preemption threshold Π_i priority. The preempting task τ_k may only preempt the currently executing task τ_i if τ_k has a nominal priority p_k exceeding the assigned preemption threshold Π_i.

Fixed PPP algorithms sought to minimize preemption overhead in fixed priority task sets in early work by Simonson and Patel [20] and Lee et al. [8]. For tasks executing via preemption triggered floating non-preemptive regions, Marinho et al. [21] successfully computed an upper-bound on task CRPD. Lastly, work in fixed priority (FP) preemption threshold schedulability (PTS) analysis by Bril et al. [22], supporting task sets with arbitrary deadlines, successfully assigned optimal priority thresholds by minimizing independent CRPD costs. This work was later complimented by combining the optimal threshold assignment algorithm with a simulated annealing algorithm that optimizes task layout [23]. The effectiveness of these heuristic solutions is limited by the employed cost model.

Bertogna et al. [4] [5] realized an optimal PPP algorithm with linear time complexity for strictly linear CFG structures. A pseudo-polynomial preemption point placement algorithm was realized by Peng et al. [6] supporting well structured series/parallel conditional CFG structures. The limitations of these solutions stem from utilizing the less accurate independent CRPD cost model as compared to the interdependent cost model accounting for the dependency between selected preemption points. Recently, a quadratic PPP algorithm was proposed by Cavicchio et al. [7] using the interdependent CRPD cost model for linear CFG structures.

In this paper, we extend the dynamic programming algorithm proposed by Peng et al. [6] to incorporate the more accurate interdependent CRPD cost model supporting instruction and data direct-mapped caches. Our conditional PPP algorithm realizes an enhanced minimized safe upper bound preemption point placement solution resulting in substantial improvements over previous PPP methods commensurate with the accuracy improvements using interdependent CRPD costs. Furthermore, the handling of inline function calls is refined into function definition and function invocation components providing the basis for non-inline function support. The problem of solving non-inline functions is a complex topic to be addressed in future work.

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 (G_i, C_i^{NP}, D_i, T_i) where G_i is a series/parallel graph describing the real-time task code, 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 $D_i \le T_i$. The system utilizes a preemptive scheduler with each task/job containing $N_i + 1$ number of basic blocks denoted (δ_i^0, δ_i^1, δ_i^2, ..., $\delta_i^{N_i}$). We introduce the basic block notation δ_i^j where i is the task identifier and j is the basic block identifier. 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 model, a basic block is a set of one or more instructions that execute non-preemptively. Basic blocks are essentially the vertices V_i of a conditional control flow graph (CFG) connected by directed edges E_i representing the execution sequence of one or more job instructions. Figure 1 shown below illustrates the conditional basic block connection structure. The CFG for task τ_i

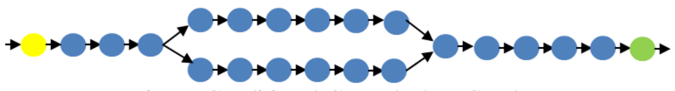


Fig. 1: Conditional Control Flow Graph.

denoted G_i is a tuple (V_i,E_i,δ_i^s,δ_i^e) representing the real-time task code structure. Flow graphs are directed graphs describing the task execution sequence from a starting basic block δ_i^s to an ending basic block δ_i^e through one or more execution paths $p \in P_i(G_i, \delta_i^s, \delta_i^e)$, where $P_i(\cdot)$ denotes all possible execution paths starting with δ_i^s and ending at δ_i^e in G_i. An execution path p through G_i is an ordered sequence of basic blocks from some starting instruction δ_i^s to an ending instruction δ_i^e. A directed edge $e_i^{x,y} = (\delta_i^x, \delta_i^y)$ where $e_i^{x,y} \in E_i$ connecting basic blocks δ_i^x and δ_i^y represents the execution of basic block δ_i^x immediately preceding the execution of basic block δ_i^y. We introduce an operator $\preceq_p$ describing a more general execution precedence relation, where $\delta_i^x \preceq_p \delta_i^y$ represents the execution of basic block δ_i^x preceding the execution of basic block δ_i^y by zero or more instructions along some path p. We further introduce notation describing a subgraph in task τ_i as G_i^a with the tuple (V_i^a,E_i^a,$\delta_i^{s^a}$,$\delta_i^{e^a}$) where a is the subgraph identifier, $\delta_i^{s^a}$ is the starting basic block, and $\delta_i^{e^a}$ is the ending basic block. Similar notation used to describe the paths in subgraph a is given as $P_i^a(G_i^a, \delta_i^{s^a}, \delta_i^{e^a})$. Preemptions are permitted at the edges $e_i^{x,y}$ between basic blocks. We introduce 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

$$C_i^{NP} = \max_{p \in P_i(G_i, \delta_i^s, \delta_i^e)} [\Sigma_{\delta_i^j \in p}\ b_i^j]. \quad (1)$$

The interdependent CRPD preemption cost, denoted by ξ_i as shown in Equation 5, is a function of the current and next preemption points and derived from the instruction and data loaded cache blocks (LCBs) for direct mapped caches as proposed in [7].

$$\begin{aligned} &\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}) = \\ &[\mathit{UCB}(\delta_i^{curr})\ \cap [\cup_{\nu \in \lambda} \mathit{AUCB}(\delta_i^{\nu})]] \cap [\cup_{\tau_k \in hp(i)} \mathit{ECB}(\tau_k)] \end{aligned} \quad (2)$$

$$\text{where } \lambda \overset{\text{def}}{=} \{\nu | \nu \in p; p \in P_i(G_i, \delta_i^{curr}, \delta_i^{next})\}$$

and the accessed UCBs, denoted AUCBs is given by:

$$\mathit{AUCB}(\delta_i^j) = \mathit{UCB}(\delta_i^j) \cap \mathit{ECB}(\delta_i^j). \quad (3)$$

A memory block m is called a useful cache block (UCB) 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 [8] [10]. A memory block m of the preempting task is called an evicting cache block (ECB), if it may be accessed during the execution of the preempting task [8] [10]. Once we have the set of cache blocks that must be re-loaded due to preemption, the CRPD related preemption overhead is given by:

$$\gamma_i(\delta_i^{curr}, \delta_i^{next}) = |\mathit{LCB}(\delta_i^{curr}, \delta_i^{next})| \cdot \mathit{BRT}. \quad (4)$$

where BRT is the cache block reload time; and $\mathit{LCB}(\delta_i^{curr}, \delta_i^{next})$ 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 [7].

$$\begin{aligned} \xi_i(\delta_i^{curr}, \delta_i^{next}) \ &= \gamma_i(\delta_i^{curr}, \delta_i^{next}) + \pi + \sigma + \\ &\quad \eta(\gamma_i(\delta_i^{curr}, \delta_i^{next})). \end{aligned} \quad (5)$$

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]. 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 [18]) has used the above information to determine the value of Q_i for each task. The determination of Q_i is dependent on the placement of preemption points. Therefore, we assume that Q_i is provided by such an approach.

IV. Real-Time Conditional Flow Graph

The types of real-time conditional flow graphs used in our work belong to the class of graphs known as series-parallel flow graphs [24]. We use the series/parallel terminology to describe the supported graph composition steps. Series-parallel flow graphs can be created using varying sequences of three basic operations, namely graph creation, series composition, and parallel composition. Creation of graph G_i^A consists of two basic blocks as vertices, δ_i^s and δ_i^e, containing a connecting directed edge $e_i^{s,e} = (\delta_i^s, \delta_i^e)$ as shown in Figure 2. Series composition takes two disjoint graphs, G_i^A and G_i^B with a connecting directed edge $e_i^{e^a,s^b} = (\delta_i^{e^a}, \delta_i^{s^b})$ where $\delta_i^{e^a}$ represents the sink node of graph G_i^A and $\delta_i^{s^b}$ represents the source node of graph G_i^B as shown in Figure 3.

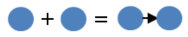


Fig. 2: Graph Creation.

Parallel composition takes two disjoint graphs, G_i^A and G_i^B, and two new basic blocks, δ_i^s and δ_i^e, with edges $e_i^{s,s^a} = (\delta_i^s, \delta_i^{s^a})$, $e_i^{s,s^b} = (\delta_i^s, \delta_i^{s^b})$, $e_i^{e^a,e} = (\delta_i^{e^a}, \delta_i^e)$, $e_i^{e^b,e} = (\delta_i^{e^b}, \delta_i^e)$ where $\delta_i^{s^a}$ and $\delta_i^{s^b}$ represent the source nodes, and $\delta_i^{e^a}$ and $\delta_i^{e^b}$ represent the sink nodes of graphs G_i^A and G_i^B respectively as shown in Figure 4. Our previous work was limited to linear control flow graphs constructed using graph creation and series composition operations only. To demonstrate the applicability of series-parallel graphs to modern real-time applications, well known and commonly used real-time structured programming language constructs such as ordered linear statement sequences, if-then statements, if-then-else statements, switch statements, bounded unrolled loops, and inline functions [25] comprise the supported software artifacts. One can readily observe that series-parallel graphs are partitioned into series and parallel connected linear code sections of basic blocks each having single entry and exit points. Due to the constrained resources of most real-time embedded processors, well written programs must employ a safe subset of programming language constructs such as conditional statements, bounded loops, and efficient functions with no recursion guaranteed to terminate within a bounded execution time in order to support cooperative tasking [26]. As a result, the real-time task code represented via series-parallel graphs can be efficiently implemented by most structured programming languages such as C. In the following sections, we present a context free graph grammar describing the series-parallel graphs supported in our work.

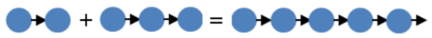


Fig. 3: Series Composition.

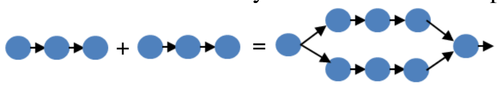


Fig. 4: Parallel Composition.

A. Grammar Background

In this section, we present a brief overview of context-free graph grammars. Historically, graph grammars have been used to facilitate the code optimization phase of program compilation [24]. In our work, we use a graph grammar to 1) recognize and construct control flow graphs conforming to the series-parallel graph structure previously described, and 2) generate intermediate structured programmatic constructs that can be efficiently solved as smaller subproblems and subsequently combined together to solve larger subproblems realizing our real-time conditional code based PPP algorithm.

Formally, a grammar $\mathcal{G}$ defines a textual language $L(\mathcal{G})$ that is parsed and recognized via a set of production rules. Production rules are of the form $LHS \leftarrow RHS$ where the left-hand side (LHS) is a non-terminal string and the right-hand side (RHS) contains non-terminal and/or terminal strings. A non-terminal string is a symbolic syntactic variable denoting some valid language construct. A terminal string represents some abstract or symbolic construct that is part of a textual based language L($\mathcal{G}$). The application of a production rule means the non-terminal string on the left-hand side is substituted for the non-terminal and/or terminal strings on the right-hand side. The process of rewriting or substituting language strings in this manner is called a derivation. In the compiler domain, the set of valid language symbols are also known as tokens. Typically, these language symbols consist of numerical strings, keywords, identifiers, or symbols, comprising a program. A grammar $\mathcal{G}$ whose production rules contain only non-terminal strings on the left-hand side of each production is called a context free grammar. A context free grammar $\mathcal{G}$ where each valid string $S \in L(\mathcal{G})$ has a unique derivation sequence that recognizes S is called unambiguous.

Like their textual counterparts, context free graph grammars consist of production rules containing both non-terminal and terminal strings. However, in a graph grammar, a terminal string represents a single vertex or basic block and a non-terminal string represents a set of vertices or basic blocks and the directed edges connecting them. Therefore, we can think of a graph grammar as a set of production rules describing how basic blocks are connected thereby representing a valid control flow graph $G \in L(\mathcal{G})$. Formally, a graph G is in the language $L(\mathcal{G})$, if there exists a sequence of derivations, starting from a specified non-terminal node, that uses the productions of $\mathcal{G}$ and results in graph G [6].

In the following subsection, we present a context free graph grammar specification describing the real-time conditional CFGs that are addressed in our work. *Real time code snippets exemplifying each grammar production rule are presented in the technical report [27].*

B. Grammar Specification

Our real-time conditional graph grammar production rules are specified in this paper in Backaus-Naur form (BNF) presented in section VI. Non-terminals are textually denoted between angle brackets ⟨CB⟩ and graphically denoted by an enclosing box. Terminals or basic blocks are textually denoted by (δ_i^j, b_i^j) where δ_i^j is the basic block identifier and b_i^j is the WCET, and graphically by a filled circle. The limitations of series-parallel graphs eliminate the use of goto statements and return statements preceding the end of a function. It is well known that real-time structured programs can be exclusively comprised of sequential statements, conditional statements, functions, and loop statements only [28].

V. Problem Statement

Using our series-parallel graph structure, the objective is to select a set of effective preemption points ρ_i that minimizes the WCET+CRPD of task τ_i whose real-time condition code is given by graph G_i, subject to the constraint that all non-preemptive regions must be less than or equal to the maximum allowable non-preemptive region parameter Q_i. The problem we solve in this paper is given by:

Problem Statement:
Given a real-time conditional graph $G_i \in \mathcal{G}$, an interdependent CRPD cost function $\xi_i(\delta_i^x, \delta_i^y)$ and WCET b_i^j for each basic block, find a set of effective preemption Points (EPPs) $\rho_i \subseteq E$ that minimizes the cost function:

$$\Phi_i(G_i, \rho_i) \overset{\text{def}}{=} \max_{p \in P_i(G_i, \delta_i^s, \delta_i^e)} \Big[\sum_{\delta_i^x \in p} b_i^x + \sum_{\substack{(\delta_i^x, \delta_i^y) \in \rho_i \\ \delta_i^x \prec_p \delta_i^y}} \xi_i(\delta_i^x, \delta_i^y) \Big] \tag{6}$$

subject to the constraint $\forall p \in P_i(G_i, \delta_i^s, \delta_i^e), \delta_i^w \in \rho_i, \exists e_i^{u,v} = (\delta_i^u, \delta_i^v), e_i^{x,y} = (\delta_i^x, \delta_i^y)$ *where* $e_i^{u,v}$, $e_i^{x,y} \in \rho_i$ ::

$$\Big[\sum_{\substack{\delta_i^x \in p \\ \delta_i^u \preceq_p \delta_i^w \preceq_p \delta_i^x}} b_i^w + \xi_i(\delta_i^u, \delta_i^x) \Big] \leq Q_i \tag{7}$$

VI. Preemption Point Placement Algorithm

In this section, we present a dynamic-programming algorithm that achieves an improved solution ρ_i to the effective PPP problem compared to existing PPP methods as outlined in section V. The set of selected feasible preemption points ρ_i minimizes our WCET cost objective function $\Phi_i(G_i, \rho_i)$ in that any other set of preemption points $\rho_i^{'}$ would result in a WCET cost $\Phi_i^{'}(G_i, \rho_i) \geq \Phi_i(G_i, \rho_i)$. To sufficiently describe our dynamic-programming algorithm, we first present a motivating example, then a high-level overview, followed by a recursive formulation based on our real-time conditional context-free grammar $\mathcal{G}$. The production rules described in the recursive formulation are applied to the CFG as part of the individual parsing steps in a bottom-up fashion.

A. Motivating Example

To present the benefits of preemption point placement using the interdependent CRPD model, consider the following example as shown in Figure 5. The WCET costs

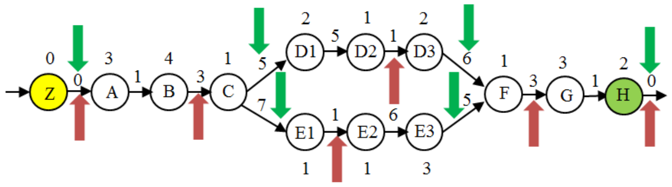
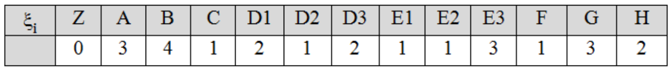


Fig. 5: Motivating Example.

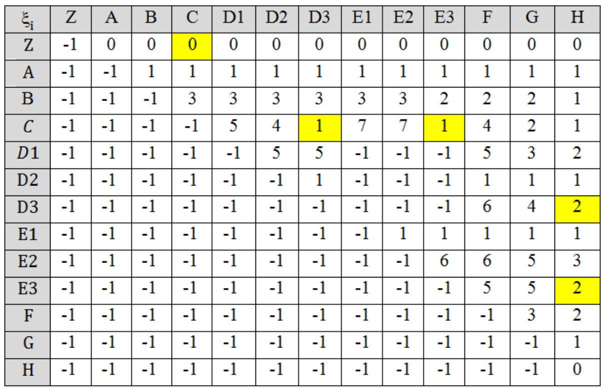
are given for each basic block along with the independent CRPD costs shown along each edge between adjacent basic blocks and summarized in Figure 6. The interdependent



ξ_i	Z	A	B	C	D1	D2	D3	E1	E2	E3	F	G	H
	0	3	4	1	2	1	2	1	1	3	1	3	2

Fig. 6: Independent CRPD Costs.

CRPD cost matrix summarizes the CRPD costs for each pair of connected basic blocks illustrating the opportunities for cost reduction as shown in Figure 7. Unconnected basic



ξ_i	Z	A	B	C	D1	D2	D3	E1	E2	E3	F	G	H
Z	-1	0	0	0	0	0	0	0	0	0	0	0	0
A	-1	-1	1	1	1	1	1	1	1	1	1	1	1
B	-1	-1	-1	3	3	3	3	3	3	2	2	2	1
C	-1	-1	-1	-1	5	4	1	7	7	1	4	2	1
*D*1	-1	-1	-1	-1	-1	5	5	-1	-1	-1	5	3	2
D2	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	1	1	1
D3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	6	4	2
E1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1
E2	-1	-1	-1	-1	-1	-1	-1	-1	-1	6	6	5	3
E3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	5	2
F	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	3	2
G	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1
H	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Fig. 7: Interdependent CRPD Costs.

blocks have -1 CRPD cost entries. The upward pointing arrows denote the minimum independent CRPD cost solution whose WCET and preemption cost is 26. The downward pointing arrows denote the minimum interdepedent CRPD cost solution whose WCET and preemption cost is 22 for both paths. The interdependent PPP algorithm chooses alternate preemption points C, $D3$, and $E3$ in accordance with the reduced preemption cost at those locations thereby illustrating the benefits of the interdependent CRPD model as highlighted in Figure 7.

B. High-Level Overview

Dynamic programming algorithms are used to efficiently implement complex algorithms where solutions to smaller subproblems are computed, stored, and subsequently reused in the solutions to larger subproblems. In our approach, we compute solutions to subsets of the real-time conditional control flow graph as it is being constructed in accordance with our grammar $\mathcal{G}$ production rules. Grammar $\mathcal{G}$ is structured in such a way that solutions are computed in the following order at each level of the parse tree, namely, 1) basic blocks, 2) linear sections, 3) conditional sections, and 4) aggregate block structures. We use the maximum non-preemptive region parameter Q_i as a suitable constraint on the number of computed solutions stored for each subgraph. We introduce two solution interface parameters, ζ_{pred}, and ζ_{succ}, denoting the non-preemptive execution times that a given solution presents to predecessor and successor subgraphs when subgraphs are combined to form solutions to larger subgraphs. To illustrate this concept, consider the following intermediate graph structure G_i^A with a proposed set of preemption points selected as denoted by the up and down arrows shown in Figure 8. Alternatively, we can visualize the

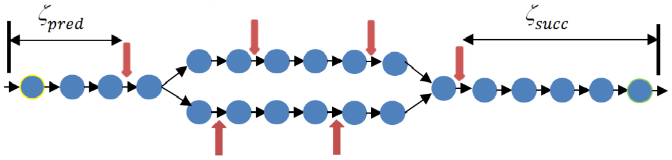


Fig. 8: Subgraph Solution Interface.

solution interface parameters ζ_{pred} and ζ_{succ} as basic blocks whose execution times are ζ_{pred} units and ζ_{succ} units respectively as shown in Figure 9. In this simplistic model, we note that preemption is not permitted at the exterior edges as the added basic blocks denote non-preemptive execution. We use infinite weight edges to enforce non-preemption between some basic blocks. Therefore, for each subgraph, we must

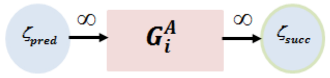


Fig. 9: Equivalent Subgraph Solution Interface.

compute at most Q_i^2 distinct solutions for each value of ζ_{pred} and ζ_{succ} in the range of $[0 \ldots Q_i - 1]$. We can think of the WCET cost and associated preemption point solutions as a set of $Q_i \times Q_i$ matrices, denoted as $\Phi_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ and $\rho_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ respectively. Later, when subgraph G_i^A is combined with other programmatic constructs in the parse tree to solve a larger subproblem, we use the ζ_{pred} and ζ_{succ} parameters to constrain which solutions from each subgraph can be combined and considered as potential solutions for the larger subgraph. We introduce two functions used to identify the visible predecessor preemption points, denoted ρ_i^{pred} and the visible successor preemption points, denoted ρ_i^{succ}. The function $\rho_i^{pred}(G_i^A, \zeta_{pred}, \zeta_{succ})$ returns the set of visible preemption points in the intermediate solution that may be reached along any path $p \in P_i(G_i^A, \delta_i^{s^A}, \delta_i^{e^A})$ starting at the first basic block $\delta_i^{s^A}$ and reaching some basic block $\delta_i^y \in \rho_i^{pred}$ by executing non-preemptively subject to the constraint that $\zeta_{pred} \leq Q_i$. Similarly, the function $\rho_i^{succ}(G_i^A, \zeta_{pred}, \zeta_{succ})$ returns the set of selected preemption points in the intermediate solution that may be reached along any path $p \in P_i(G_i^A, \delta_i^{s^A}, \delta_i^{e^A})$ starting at basic block $\delta_i^y \in \rho_i^{succ}$ and reaching the ending basic block $\delta_i^{e^A}$ by executing non-preemptively subject to the constraint that $\zeta_{succ} \leq Q_i$. Thus, the sets ρ_i^{pred} and ρ_i^{succ} are used determine the additive preemption cost between two subgraphs whose solutions are being combined. In the next subsection, we present a recursive formulation that achieves an effective minimized safe upper bound preemption solution for each larger subgraph as a combination of the minimized safe upper bound preemption solutions to the respective smaller subgraphs.

C. Recursive Formulation

In accordance with our context-free grammar $\mathcal{G}$, the various subgraphs G_i^A are created via applying the production rules on a textual based graph description that conforms to the language $L(\mathcal{G})$ presented below. As each production rule or derivation is applied, a defined subset of basic blocks and their connection relationships are assigned to subgraphs denoted by their non-terminal symbol. For instance, the subgraph created by the linear blocks production is denoted with a suitable abbreviation to establish a proper association between the production rule, the subgraph G_i^{LB}, the WCET cost objective function Φ_i^{LB}, and the set of selected preemption points ρ_i^{LB}.

We now present the grammar $\mathcal{G}$ production rules focusing on the computation of the WCET cost objective function $\Phi_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ and the associated set of selected preemption points $\rho_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ comprising the set of solutions generated at each level of the parse tree for all values of ζ_{pred} and ζ_{succ} in the range of $[0 \ldots Q_i - 1]$. In this section, we present production rules a) and b) supporting the conditional CFG structures, while the remaining production rules c) through k) containing the other programming constructs such as blocks, loops, and functions are presented in the appendix due to space constraints.

For **instruction production rule (a)**, we have:

$$<SB> \rightarrow (\delta_i^j, b_i^j)$$

The derivation of production rule (a) creates a subgraph G_i^{SB} containing a single basic block, δ_i^j. The associated WCET cost and preemption point functions are given by:

$$\Phi_i^{SB(j)}(\zeta_{pred}, \zeta_{succ}) = \left\{ \begin{array}{ll} \infty, & \text{if } b_i^j > Q_i \\ b_i^j, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) \leq Q_i \\ b_i^j, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) > Q_i \end{array} \right\} \tag{8}$$

$$\rho_i^{SB(j)}(\zeta_{pred}, \zeta_{succ}) = \left\{ \begin{array}{ll} \emptyset, & \text{if } b_i^j > Q_i \\ \emptyset, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) \leq Q_i \\ \delta_i^j, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) > Q_i \end{array} \right\} \tag{9}$$

For **conditional production rule (b)**, we have:

$$<CB> \rightarrow <SB> \;\; [\; <Blocks> \;] \; [\; <Blocks> \;]+ \;\; <SB>$$

The derivation of production rule (b) creates a subgraph G_i^{CB} concatenating a single basic block, δ_i^j in subgraph $G_i^{SB(j)}$, followed by one or more blocks each forming a conditional section in subgraph $G_i^{CS_a}$ where $a \in [1, r]$, ending with a single basic block, δ_i^k in subgraph $G_i^{SB(k)}$.

Solutions previously computed for the r conditional sections are combined with the solutions computed for the leading and trailing basic blocks δ_i^j and δ_i^k respectively. Production rule (b) exhibits time complexity executing in $O(N_i log(N_i) r Q_i^2)$ time. Each <*SB*> contains 2 solutions with each of the r <*Blocks*> structures containing Q_i^2 solutions. The associated WCET[1] cost function is given by:

$$\Phi_i^{CB}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \max_{a \in \mathbb{N}:1 \le a \le r} \Big\{ \min_{s,t,u} \Big\{ \Phi_i^{SB(j)}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_s}) + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t}) + max_{\delta_i^v, \delta_i^w}[\xi_i(\delta_i^v, \delta_i^w)] + \Phi_i^{SB(k)}(\zeta_{pred_u}, \boldsymbol{\zeta_{succ}}) \Big\} \Big\} \tag{10}$$

where the variables in the min and max expressions (ζ_{succ_s}, ζ_{pred_t}, ζ_{succ_t}, ζ_{pred_u}, δ_i^m, δ_i^n, δ_i^v, and δ_i^w) represent values where the function $\Phi_i^{CB}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}})$ is minimized subject to the following constraints:

$$\begin{aligned} (\zeta_{succ_s} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_t}) &\le Q_i \\ (\zeta_{succ_t} + max_{\delta_i^v, \delta_i^w}[\xi_i(\delta_i^v, \delta_i^w)] + \zeta_{pred_u}) &\le Q_i \\ \delta_i^m &\in \rho_i^{succ}(G_i^{SB(j)}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_s}) \\ \delta_i^n &\in \rho_i^{pred}(G_i^{CS_a}, \zeta_{pred_t}, \zeta_{succ_t}) \\ \delta_i^v &\in \rho_i^{succ}(G_i^{CS_a}, \zeta_{pred_t}, \zeta_{succ_t}) \\ \delta_i^w &\in \rho_i^{pred}(G_i^{SB(k)}, \zeta_{pred_u}, \boldsymbol{\zeta_{succ}}) \end{aligned} \tag{11}$$

The associated preemption point function is given by:

$$\rho_i^{CB}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \rho_i^{SB(j)}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_s}) \bigcup_{a=1}^{r} \rho_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t}) \bigcup \rho_i^{SB(k)}(\zeta_{pred_u}, \boldsymbol{\zeta_{succ}}) \tag{12}$$

The basic blocks δ_i^m, δ_i^n, δ_i^v, and δ_i^w represent elements of the predecessor and successor visible preemption sets used to determine the interdependent preemption cost of the combined solutions. It is important to note that in order to maintain a safe cost bound, we must use the cost associated with the worst-case predecessor/successor preemption points for each solution when smaller solutions are combined into larger solutions. Formally, given the Φ_i and ρ_i functions for each substructure of G^A where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before (resp., ζ_{succ}) after and Φ_i^A represents a safe upper bound on the total WCET and preemption cost of that solution. Thus, the solutions selected by our algorithm are minimized in accordance with our cost functions, however, our use of the maximum preemption cost when combining solutions potentially destroys global optimality. This concept applies to all presented production rules.

[1]ζ_{pred} and ζ_{succ} are bolded to denote that they are constants in determining the costs for Equations 10-12; all other ζ and δ values vary over their respective ranges.

Theorem 1. *Given Φ_i and ρ_i functions for each substructure of CB where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production (b) over a feasible G_i, G_i^{CB} and Q_i results in a feasible solution ρ_i^{CB} and a safe upper bound Φ_i^{CB} given by Equations 10, 11, and 12 respectively.*

Proof: The proof is by direct argument. We need to prove that our solution ensures that the task level Q_i constraint is not violated and the cost function $\Phi_i^{CB}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound. To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{SB(j)}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \ldots Q_i]$ represent a safe upper bound cost solution for subgraph $G_i^{SB(j)}$ for basic block δ_i^j, with its corresponding set of selected preemption points denoted by $\rho_i^{SB(j)}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for basic block δ_i^j. We make an identical statement for subgraph $G_i^{SB(k)}$ for basic block δ_i^k, whose cost function is denoted $\Phi_i^{SB(k)}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{SB(k)}(\zeta_{pred_u}, \zeta_{succ})$. We make a similar statement for subgraph $G_i^{CS_a}$ starting at basic block $\delta_i^{scs_a}$ and ending at basic block $\delta_i^{ecs_a}$, whose cost function is denoted $\Phi_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t})$, and whose set of selected preemption points are denoted $\rho_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t})$ where $a \in [1, r]$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{CB}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 10 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{SB(j)} \bigcup_{a=1}^{r} G_i^{CS_a} \cup G_i^{SB(k)}$ starting at basic block δ_i^j, and ending at basic block δ_i^k with its corresponding selected preemption points denoted by $\rho_i^{CB}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 12. Condition 1 is met in accordance with Equation 11 whose purpose is to ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range is given by $[0 \ldots Q_i - 1]$. Thus, the problem finds a feasible safe upper bound cost preemption points solution when applying production (b). ■

D. Interdependent CRPD Solution Handling

One of the primary motivations for our work is the interdependent CRPD cost model, which necessitates a series of modifications to the conditional PPP algorithm for proper solution handling. One challenge interdependent CRPD presents (that independent CRPD does not) is the preemption cost cannot be determined when the preemption solutions for basic blocks are processed using production rule (a) since the successor preemption is not known. Interdependent CRPD costs may only be determined for preemption pairs in contrast to independent CRPD, a function of a single preemption location only. This means we have to determine the maximum preemption cost for pairs of solutions that are combined as higher level block structures are processed. To accomplish this, we must have a way of determining the set of preemption points that are visible externally to adjacent solutions in order to compute the maximum preemption cost of the combined solutions. A preemption point has external visibility to adjacent blocks if there exists a path from the starting or ending section block to the preemption point with no intervening preemption points encountered. Determining the visible preemption points while combining preemption solutions adds an $O(N_i)$ factor to the algorithm for the current block. The following Algorithm 1 illustrates the method of computing the visible predecessor preemption points for an existing solution. For the visible predecessor preemption

Algorithm 1 Visible Predecessor Preemptions

```
1: function $vis\_pred\_pps(\beta, \rho)$
2:     Preemption solution block $\beta$, preemption solution $\rho$
3:     $\rho_{prev} \leftarrow \emptyset$
4:     $\varsigma_{start} \leftarrow \beta.startSection$ $\varsigma_{end} \leftarrow \beta.endSection$
5:     $\rho_{vis} \leftarrow vis\_curr\_pred\_pps(\varsigma_{end}, \varsigma_{start}, \rho, \rho_{prev})$
6:     return $\rho_{vis}$
7: end function
```

points, we start with the ending section of the block and work our way backwards towards the starting section of the block. As we move backward, the preemption points encountered replace those in the current set if all sections have preemption points. If not, then the existing preemption points are still visible and copied to the preemption set for the next iteration. This is shown in Algorithm 2. For each section block processed, the first successor preemption point encountered moving from right to left is added to the preemption point set as shown in Algorithm 3. Computing the visible successor preemption points works in an identically symmetric way using three similar algorithms.

Algorithm 2 Visible Current Predecessor Preemptions

```
1: function $vis\_curr\_pred\_pps(\varsigma_{curr}, \varsigma_{start}, \rho, \rho_{prev})$
2:     $\alpha_{sections} \leftarrow true$ $\rho_{call} \leftarrow \emptyset$ $\rho_{next} \leftarrow \emptyset$
3:     $vis\_pred\_pps\_sect(\varsigma_{curr}, \varsigma_{start}, \rho_{prev})$
4:     if $\rho_{prev} = \emptyset$ then
5:         $\rho_{call} \leftarrow \rho_{call} \cup \rho_{prev}$
6:     end if
7:     if $\varsigma_{curr} \neq \varsigma_{start}$ then
8:         for $\varsigma_{next} \in \varsigma^{pred}(\varsigma_{curr})$ do
9:             $\rho_{sect} \leftarrow vis\_succ\_pps\_sect(\varsigma_{next}, \varsigma_{start}, \rho, \rho_{call})$
10:            if $\rho_{sect} = \emptyset$ then
11:                $\alpha_{sections} \leftarrow false$
12:            else
13:                $\rho_{next} \leftarrow \rho_{next} \cup \rho_{sect}$
14:            end if
15:        end for
16:        if $\alpha_{sections} = false$ then
17:            $\rho_{next} \leftarrow \rho_{next} \cup \rho_{call}$
18:        end if
19:    else
20:        $\rho_{next} \leftarrow \rho_{call}$
21:    end if
22:    return $\rho_{next}$
23: end function
```

Algorithm 3 Visible Section Successor Preemptions

```
1: function $vis\_succ\_sect\_pps(\varsigma_{curr}, \rho, \rho_{call})$
2:     $\delta_i^{left} \leftarrow \varsigma_{curr}.leftmostBB$
3:     $\delta_i^{right} \leftarrow \varsigma_{curr}.rightmostBB$
4:     for $\delta_i^{curr} \in [\delta_i^{right}, \delta_i^{left}]$ do
5:         if $\delta_i^{curr} \in \rho$ then
6:             $\rho_{call} \leftarrow \rho_{call} \cup \delta_i^{curr}$
7:             Exit the For Loop.
8:         end if
9:     end for
10:    return $\rho_{call}$
11: end function
```

VII. Evaluation

Our conditional PPP algorithm will be evaluated using two methods: 1) characterization and measurement of preemption costs using real-time application code, and 2) a breakdown utilization schedulability comparison of various PPP algorithms. Each PPP algorithm evaluated either uses an independent or interdependent CRPD cost model.

A. Preemption Cost Characterization

Our study will utilize a subset of real-time tasks from the Malardalen University (MRTC) WCET benchmark suite [29] for comparing various PPP algorithms. The task code was compiled using the GCC MIPS Cross Compiler for MIPS series processors with separate instruction and data $1KB$ direct-mapped caches.

The compiled real-time task code was processed by the Heptane Static WCET analysis tool [30]. Heptane is used to determine the set of section blocks, the section block WCETs, the CFG structure, and the cache state at each instruction by analyzing the program executable. The analysis results are then imported into a Java benchmark parser program, designed to generate the task code grammar used by our conditional PPP algorithm. To accomplish this, high level programming constructs such as loops, conditionals, functions, and block statements must be recognized from the low level compilation output.

Once the benchmark CFG structure has been constructed, the results of the Heptane cache state analysis are imported and used to compute the instruction and data UCBs ($\Upsilon_I(\delta_i^j)$ and $\Upsilon_D(\delta_i^j)$ respectively) at each program location. These sets are then used to compute the shared LCBs, along with the interdependent preemption cost matrix.

The intersection of the cache state snapshots from δ_i^j to δ_i^k are used to calculate shared LCBs. Shared LCBs represent the set of cache lines whose contents remain unevicted after execution of basic blocks $\{\delta_i^{j+1}, \delta_i^{j+2}, ..., \delta_i^k\}$. As such, shared LCBs will continue to be present in the cache prior to the execution of basic block δ_i^{k+1}. Thus, a safe upper bound on the LCBs shared between each basic block pair can be represented by the set of unchanged cache lines. The following equation below formalizes this computation where the instruction cache snapshots are denoted $\Upsilon_I(\delta_i^j)$ and the data cache snapshots denoted $\Upsilon_D(\delta_i^j)$.

$$LCB(\delta_i^j, \delta_i^k) \subseteq \bigcap_{m=j+1}^{k} \Upsilon_I(\delta_i^m) \cup \Upsilon_D(\delta_i^m) \qquad (13)$$

1) Availability: The following tools and data sets may be used to verify and reproduce our work. The MIPS GCC cross compiler and the Heptane static worst case execution time tool are freely available. The research community may reproduce and leverage our work via the developed programs and analyzed data archived at GitHub [31].

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. [5] and Peng et al. [6]). In terms of LCB computation this implies that the maximum LCB value over all subsequent program points must be used as the CRPD cost:

$$\max\{LCB(\delta_i^j, \delta_i^k) \mid \delta_i^j \preceq \delta_i^k\} \qquad (14)$$

The interdependent CRPD approach, representing CRPD cost for pairs of preemptions, is illustrated in the following graphs. The graph lines shown characterize the minimum and maximum shared LCBs between program points. The x-axis represents the first program preemption point, denoted δ_i^j. The y-axis measures the shared LCB count with the secondary program point, denoted δ_i^k, annotated at each graph point as shown. The first graph shown in Figure 10 characterizes the instruction cache CRPD costs for the FFT benchmark program. Each point on both curves plots the minimum or maximum CRPD value for the first preemption point given by the x-axis, and the next preemption point annotated on the graph. At program point δ_i^1, the minimum CRPD value is coupled with program point δ_i^8 having a shared LCB count of 175 whereas the single-valued CRPD computation method finds 250 shared LCBs coupled at program point δ_i^5. Comparison of the dual-valued interdependent CRPD with the single-valued independent CRPD methods can be visualized by comparing the vertical distance between the minimum and maximum CRPD curves of Figures 10, 11, or 12. The maximum CRPD cost at any program location represents a mandatory safe value for any single-valued independent CRPD approach. In contrast, our interdependent CRPD method offers the potential minimum

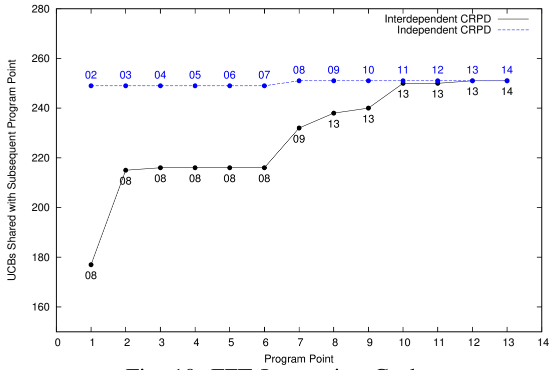


Fig. 10: FFT Instruction Cache.

value reported on the solid line. The benefit provided in considering location aware interdependent CRPD cost is captured by the difference between the minimum and maximum CRPD cost curves fueling the improved performance of our conditional PPP algorithm. The variability in the minimum and maximum CRPD costs further exemplifies the benefits as illustrated in the second and third graphs, representing the lms benchmark task instruction cache in Figure 11 and the cover benchmark task instruction cache in Figure 12 respectively. In this paper, we have presented the variability witnessed in the instruction cache graphs, as the conditional CFG structure emphasizes the instruction cache effect on CRPD. Maximum and minimum instruction and data cache costs for the other MRTC tasks exhibit similar variability.

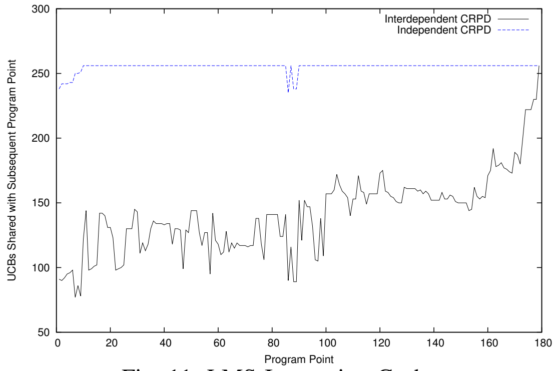


Fig. 11: LMS Instruction Cache.

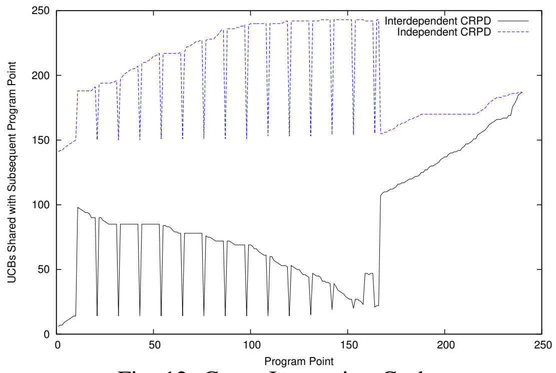


Fig. 12: Cover Instruction Cache.

Review of the instruction and data cache graphs led to some notable observations. The maximum and minimum

LCBs converged towards the end of each tasks CFG due to the decreasing CFG structure remaining thereby reducing the LCB count variability. Downward spikes are well aligned with task block boundaries, such as loops, conditionals, and functions. Early upward trends result from task initialization code. The separation between the two curves illustrates the accuracy improvement of our interdependent CRPD method versus independent CRPD methods. Our proposed PPP algorithm utilizes the more accurate interdependent CRPD cost leading to substantial schedulability improvements.

B. *Breakdown Utilization*

Now that the reduced task preemption overhead benefits of the more precise interdependent CRPD cost model has been presented, we turn our attention to the benefits in task set schedulability. To evaluate task set schedulability, breakdown utilization performance was compared for several PPP algorithms on selected MRTC benchmark [29] tasks. The goal of breakdown utilization analysis [32] is to determine the utilization at which a task set becomes unschedulable. The PPP algorithms compared in our study include the Bertogna et al. PPP algorithm [5] (BEPP), our linear PPP algorithm [7] (LEPP), the Peng et al. conditional PPP algorithm [6] (PEPP), and our proposed conditional PPP algorithm (CEPP).

The detailed steps of our iterative schedulability algorithm integrated with our conditional PPP algorithm are described in Algorithm 4. Task set utilization, given by U, is controlled by setting each tasks deadline and period to $D_i = T_i = u \cdot C_i^{NP}$. The constant, u, is binary search incremented in small steps until the task set becomes schedulable. Then u is binary search decremented in small steps until the task set becomes unschedulable, resulting in the breakdown utilization U_B. For $BEPP$ and $PEPP$, the maximum shared LCB counts previously obtained form the independent CRPD input. Lastly, for $LEPP$ and $CEPP$, the explicit shared LCB counts previously obtained comprise the interdependent CRPD input for our linear and conditional explicit PPP algorithms. The remaining input variables required by the breakdown utilization algortithm are C_i^{NP} and BRT. C_i^{NP} was computed as the maximum number of cycles to complete task execution non-preemptively. During each run of the breakdown utilization study, the BRT parameter is swept from 1 ns to 640 ns per MIPS processor family performance. Due to compiler optimizations, we had to post-process the MRTC tasks to apply our grammar. Ideally, an automated tool would exist for this step; however, such a tool is beyond the scope of this paper due to the explicit compiler-level detail required. For this paper, we manually post-processed the following MRTC tasks: simple, bs, fibcall, lcdnum, sqrt, qurt, insertsort, ns, ud, crc, expint, jfdctint, matmult, and bsort100 [31]. Using the completed MRTC tasks, the breakdown utilization comparison between various PPP methods is summarized in Figure 13. The breakdown utilization results indicate that the $CEPP$ (resp. $LEPP$) algorithm dominates the $PEPP$ (resp. $BEPP$) algorithm primarily due to the benefits of interdependent versus independent CRPD. Both the $CEPP$ and $PEPP$ algorithms dominate $LEPP$ and $BEPP$ algorithms due to the enhanced granularity of the conditional CFGs, offering more possible preemptions than the linear CFGs. As expected, the breakdown utilization values converge for each distinct graph structure (e.g. linear, conditional) as cache-overhead becomes negligible for small BRT values.

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 PPP Algorithm
 5:     if the task system is feasible/schedulable then
 6:         Increase U by decreasing $T_i$ values via binary search.
 7:     else
 8:         Decrease U by increasing $T_i$ values via binary search.
 9:     end if
10: until the utilization change is less than some tolerance.
11: The breakdown utilization is given by U.
```

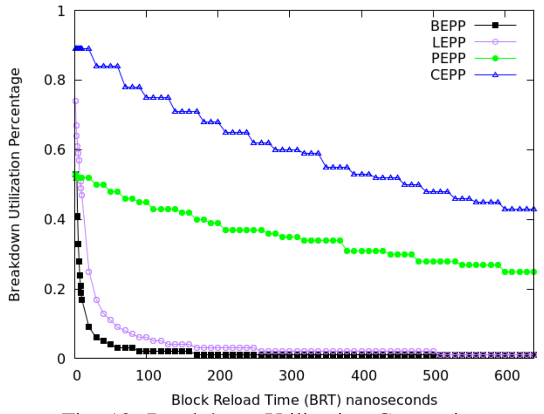


Fig. 13: Breakdown Utilization Comparison.

VIII. Conclusion

In this paper, we presented a conditional PPP algorithm using a more precise interdependent CRPD metric. By extending the interdependent CRPD cost to conditional CFG structures, further reductions in task preemption overhead were realized, leading to substantial schedulability improvements. These improvements were achieved by integrating our conditional interdependent CRPD PPP algorithm with algorithms from well established task set schedulability theory. Our iterative schedulability algorithm demonstrates the convergence of selecting preemptions balanced by the task maximum non-preemptive execution region constraint Q_i. Our experiments demonstrated improved schedulability on real-time code using interdependent CRPD.

In future work, we plan to 1) extend the breakdown utilization analysis to the remaining MRTC benchmark tasks, 2) perform a timing analysis of various PPP algorithms, 3) add support for non-inline functions, and 4) extend the techniques described here to set-associative caches.

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

[5] 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*.

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

[7] J. Cavicchio, C. Tessler, and N. Fisher, "Minimizing cache overhead via loaded cache blocks and preemption placement," *In ECRTS, 2015*.

[8] 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," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700–713, 1998.

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

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

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

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

[13] 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.

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

[15] S. Altmeyer, R. Davis, and C. Maiza, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *In RTSS, 2011*.

[16] ——, "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," *Real Time Systems, Springer*, 2012.

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

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

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

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

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

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

[23] R. Bril, S. Altmeyer, M. van den Heuvel, R. Davis, and M. Behnam, "Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation," *Real-Time Systems*, vol. 53, no. 4, pp. 403–466, Jul 2017.

[24] K. Kennedy and L. Zucconi, "Applications of a graph grammar for program control flow analysis," *In Proceedings ACM SIGACTSIGPLAN Symposium on Principles of programming languages, 1977*.

[25] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools Second Edition*, 2007.

[26] R. Wilhelm, "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS), 2008, ACM*.

[27] Technical Report. [Online]. Available: https://github.com/jcavicchio/crpd_appp/tree/master/tech_report

[28] C. Bohm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules," *Communications of the ACM*, vol. 9, no. 5, pp. 366–371, May 1966.

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

[30] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," *In WCET, 2017*.

[31] Paper Programs and Data Repository. [Online]. Available: https://github.com/jcavicchio/crpd_appp/tree/master

[32] 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 following section presents the remaining production rules for conditional structures.

A. Conditional Structures

For **blocks production rule (c)**, we have:

<Blocks> $\rightarrow$ [*<SB>*]

The derivation of production rule (c) creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{SB}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \Phi_i^{SB}(\zeta_{pred}, \zeta_{succ}) \tag{14}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{SB}(\zeta_{pred}, \zeta_{succ}) \tag{15}$$

For **blocks production rule (d)**, we have:

<Blocks> $\rightarrow$ [*<CB>*]

The derivation of production rule (d) creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{CB}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \Phi_i^{CB}(\zeta_{pred}, \zeta_{succ}) \tag{16}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{CB}(\zeta_{pred}, \zeta_{succ}) \tag{17}$$

For **aggregate blocks production rule (e)**, we have:

<Blocks> $\rightarrow$ *<SB>* *<Blocks>*

The derivation of production rule (e) creates a subgraph $G_i^{BLKS'}$ concatenating a previously created aggregate blocks basic block subgraph G_i^{SB} in series with a previously created subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\begin{aligned}\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ}) = min_{r,s}\{(\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ_r}) + \\ \max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \\ \Phi_i^{SB}(\zeta_{pred_s}, \zeta_{succ})\}\end{aligned} \tag{18}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solutions are subject to the following constraints:

$$\begin{aligned}(\zeta_{succ_r} + \max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \\ \delta_i^m \in \rho_i^{succ}(G_i^{BLKS}, \zeta_{pred}, \zeta_{succ_r}) \\ \delta_i^n \in \rho_i^{pred}(G_i^{SB}, \zeta_{pred_s}, \zeta_{succ})\end{aligned} \tag{19}$$

The associated preemption point function is given by:

$$\begin{aligned}\rho_i^{BLKS'}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ_r}) \cup \\ \rho_i^{SB}(\zeta_{pred_s}, \zeta_{succ})\end{aligned} \tag{20}$$

Theorem 2. *Given Φ_i and ρ_i functions for each substructure of $BLKS$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production (e) over a feasible G_i, G_i^{BLKS} and Q_i results in a feasible solution ρ_i^{BLKS} and a safe upper bound Φ_i^{BLKS} given by Equations 18, 19, and 20 respectively.*

Proof: The proof is by direct argument. The proof structure and line of reasoning are identical to the proof for Theorem 1, hence we omit the details here. ■

For **aggregate blocks production rule (f)**, we have:

<Blocks> $\rightarrow$ *<CB>* *<Blocks>*

The derivation of production rule (f) creates a subgraph $G_i^{BLKS'}$ concatenating a previously created conditional block subgraph G_i^{CB} in series with a previously created aggregate blocks subgraph G_i^{BLKS}. Production rule (f) exhibits the maximum time complexity for our algorithm executing in $O(N_i log(N_i)Q_i^4)$ time. Each *<SB>* contains Q_i solutions with each *<Blocks>* and *<CB>* structure containing Q_i^2 solutions. The associated WCET cost function is given by:

$$\begin{aligned}\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ}) = min_{r,s}\{(\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ_r}) + \\ max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \\ \Phi_i^{CB}(\zeta_{pred_s}, \zeta_{succ})\}\end{aligned} \tag{21}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solution combinations are subject to the following constraints:

$$\begin{aligned}(\zeta_{succ_r} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \\ \delta_i^m \in \rho_i^{succ}(G_i^{BLKS}, \zeta_{pred}, \zeta_{succ_r}) \\ \delta_i^n \in \rho_i^{pred}(G_i^{CB}, \zeta_{pred_s}, \zeta_{succ})\end{aligned} \tag{22}$$

The associated preemption point function is given by:

$$\begin{aligned}\rho_i^{BLKS'}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ_r}) \cup \\ \rho_i^{CB}(\zeta_{pred_s}, \zeta_{succ})\end{aligned} \tag{23}$$

Theorem 3. *Given Φ_i and ρ_i functions for each substructure of $BLKS$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production (e) over a feasible G_i, G_i^{BLKS} and Q_i results in a feasible solution ρ_i^{BLKS} and a safe upper bound Φ_i^{BLKS} given by Equations 21, 22, and 23 respectively.*

Proof: The proof is by direct argument. The proof structure and line of reasoning are identical to the proof for Theorem 1, hence we omit the details here. ■

The grammar we have presented thus far are focused on the production rules for conditional structures. Non-unrolled loops and functions are structured programming constructs that are also prevalent in real-time code. We present the production rules supporting these structured programming elements in the following subsections.

B. Non Unrolled Loops

For **loops production rule (g)**, we have:

<Loop> → [*<Blocks>* *<MaxIter>*]

The derivation of production rule (g) creates a subgraph G_i^{LOOP} that is equivalent to the subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\Phi_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) = \Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \times MaxIter \tag{24}$$

where valid solution combinations are subject to the following constraints:

$$(\zeta_{succ} + \zeta_{pred}) \leq Q_i \tag{25}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \tag{26}$$

For **blocks production rule (h)**, we have:

$$\textit{<Blocks>} \rightarrow [\ \textit{<LOOP>}\ ] \tag{27}$$

The derivation of production rule (h) creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{LOOP}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \Phi_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) \tag{28}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) \tag{29}$$

C. Inline Functions

Functions are split into two grammar elements, namely, function definition and function invocation. The conditional PPP algorithm generates solutions for the function definition blocks consistently with the main task function. The generated function definition preemption solutions are combined with the function invocation preemption solutions at each graph location where the function is called.

For **function definition production rule (i)**, we have:

<Function> → [*<Blocks>* *<FunctionName>*]

The derivation of production rule (i) creates a subgraph G_i^{FUNC} that is equivalent to the subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\Phi_i^{FUNC}(\zeta_{pred}, \zeta_{succ}) = \Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \tag{30}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{FUNC}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \tag{31}$$

For **function call production rule (j)**, we have:

<FunctionCall> → [*<Blocks>* *<FunctionName>*]

The derivation of production rule (j) creates a subgraph G_i^{FCALL} that is equivalent to the subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\begin{aligned}\Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ}) = min_{r,s}\{(\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ_r}) +\\ max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] +\\ \Phi_i^{FUNC}(\zeta_{pred_s}, \zeta_{succ})\}\end{aligned} \tag{32}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solution combinations are subject to the following constraints:

$$\begin{aligned}(\zeta_{succ_r} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i\\ \delta_i^m \in \rho_i^{succ}(G_i^{BLKS}, \zeta_{pred}, \zeta_{succ_r})\\ \delta_i^n \in \rho_i^{pred}(G_i^{FUNC}, \zeta_{pred_s}, \zeta_{succ})\end{aligned} \tag{33}$$

The associated preemption point function is given by:

$$\begin{aligned}\rho_i^{FCALL}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ_r}) \cup\\ \rho_i^{FUNC}(\zeta_{pred_s}, \zeta_{succ})\end{aligned} \tag{34}$$

Theorem 4. *Given Φ_i and ρ_i functions for each substructure of $FCALL$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production (e) over a feasible G_i, G_i^{FCALL} and Q_i results in a feasible solution ρ_i^{FCALL} and a safe upper bound Φ_i^{FCALL} given by Equations 32, 33, and 34 respectively.*

Proof: The proof is by direct argument. The proof structure and line of reasoning are identical to the proof for Theorem 1, hence we omit the details here. ■

For **blocks production rule (k)**, we have:

<Blocks> → [*<FunctionCall>*]

The derivation of production rule (k) creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{FCALL}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ}) \tag{35}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{FCALL}(\zeta_{pred}, \zeta_{succ}) \tag{36}$$