Tree-Based WCET Analysis on Instrumentation Point Graphs

Adam Betts* and Guillem Bernat

Real-Time Systems Research Group
University of York
York YO10 5DD, UK
abetts@cs.york.ac.uk

Abstract

This paper presents a framework for combining low-level measurement data through high-level static analysis techniques on instrumented programs in order to generate WCET estimates, for which we introduce the instrumentation point graph (IPG). We present the notion of iteration edges, which are the most important property of the IPG from a timing analysis perspective since they allow more path-based information to be integrated into tree-based calculations on loops.

The main focus of this paper, however, is an algorithm that performs a hierarchical decomposition of an IPG into an Itree to permit tree-based WCET calculations. The Itree representation supports a novel high-level structure, the meta-loop, which enables iteration edges to be merged in the calculation stage. The timing schema required for the Itree is also presented. Finally, we outline some conclusions and future areas of interest.

1 Introduction

In real-time systems, a guarantee of precise functionality requires affirmation of both functional and temporal correctness. This latter property is adhered to by breaking the system up into a number of tasks and assigning a temporal order to the task set according to a selected scheduling algorithm, e.g. earliest deadline first (EDF). The feasibility of scheduling a task set is ascertained through schedulability analysis by analysing tasks' temporal parameters relative to the available system resources. A central parameter in this analysis is that of a task's *worst-case execution time* (WCET), which represents the maximum amount of CPU time a task requires in order to complete execution *on a particular architecture.*

The field of WCET analysis is driven by two schools of thought: *static analysis* (SA) and *measurement-based* (MB) analysis. SA produces WCET estimates by building models of both the program and the processor. Mueller [14] has modelled the effects of instruction caches for arbitrary levels of associativity. Similarly, White *et al.* [22] have proposed a method to bound worst-case data cache performance. Bate *et al.* [5] have contemplated the effects of dynamic branch predictors, whilst Li *et al.* [12] have considered modelling out-of-order execution units. In spite of this progress, correctly modelling the microprocessor is handicapped by its stringent need for *predictability*, which is compromised in the presence of such hardware speed-up features. As a result, pessimistic assumptions are formulated that lead to overestimation, but real-time hardware architects are increasingly looking towards advanced microproccesors [9]. These advancements provide the motivation for MB techniques [8, 21], which are based on testing the real system instead of building a model. An evident requirement of MB techniques is that test-generation techniques must exercise the combination of input values that leads to the actual WCET, which is not a trivial problem.

The approach we present in this paper is not an end-to-end MB approach, but a hybrid method that attempts to reduce the underestimation and overestimation incurred by SA and MB techniques, respectively. This is achieved by combining low-level MB data through high-level static techniques that utilise knowledge of the program's high-level structure in the calculation stage. For these purposes, the control flow graph (CFG) of the program under analysis is instrumented, and a flow graph is derived from the paths among instrumentation points, resulting in an instrumentation point graph (IPG). The IPG subsequently serves as a basis for tree-based [7], path-based [20] and IPET [11] calculation methods.

In this paper, we formally introduce the IPG and present its two most significant properties from a timing analysis perspective: *instruction blocks* and *iteration edges*. Using partial tracing mechanisms, instruction blocks contain context-sensitive execution of basic blocks[1], whilst iteration edges permit several (sub-)paths to contribute to the WCET of a corresponding CFG loop instead of the unique worst-case path. Therefore, more path-based information can be integrated into tree-based calculations on the IPG. However, the main contribution of this paper is how to decompose an IPG into a hierarchical form, the Itree, for the purposes of tree-based calculations. The Itree representation supports a novel construct, the *meta-loop*, which augments tree-based calculations by combining iteration edges and their respective loop body in the calculation stage. We also give the timing schema required for an Itree.

In the next section we outline some graph terminology to be used in the remainder of the paper. In section 3, we discuss a taxonomy of instrumentation profiles and how these relate to calculation techniques performed on the IPG. In section 4 we introduce the IPG and discuss instruction

*This research is sponsored, in part, by the EPSRC funded project, DIRC.

[1] A basic block is a sequence of consecutive instructions such that flow of control can only enter at the beginning and leave at the end [2]

blocks and iteration edges. In section 5, we give a detailed description of the algorithm that decomposes an IPG into an Itree. We also discuss the properties of the Itree, including the meta-loop construct, and the timing schema. Finally, we present some conclusions and future areas of work.

2 Graph Terminology and Notation

A *graph* $G = (N, E)$ is a pair of finite sets N and E, called nodes and edges respectively, where $E \subseteq N \times N$. A *directed edge* $x \rightarrow y$, $x, y \in N$, is written (x, y), where x and y are called the *source* and the *destination* of the edge, respectively. Further, x is an *immediate predecessor* of y and y is an *immediate successor* of x. The immediate predecessors and successors of a node $n \in N$ shall be denoted $pred(n)$ and $succ(n)$, respectively. A *branch node* b has $|succ(b)| > 1$, whilst a *merge* node c has $|pred(c)| > 1$. A *directed graph* (*digraph*) $H = (N, E)$ is a graph where each $e \in E$ is a directed edge. A *path* of length m is a sequence of edges $(x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_{m+1})$, where each $x_i \rightarrow x_{i+1} \in E$. The notation $x_1 \stackrel{*}{\rightarrow} x_n$ denotes a path of length zero or more, whereas $x_1 \stackrel{+}{\rightarrow} x_n$ denotes a path of length one or more.

A graph is *connected* if there is a non-empty path between any distinct pair of nodes $\{x, y\}$. A digraph is *weakly* connected if its underlying undirected graph is connected. A control flow graph is a weakly connected, digraph $C = (N, E, start, exit)$, $start, exit \in N$, where we assume that $start$ has no predecessors, $exit$ has no successors and that $start \rightarrow exit \in E$. Furthermore, it is assumed for every $n \in N$ there is a directed path from $start$ to n and a directed path from n to $exit$, i.e. no dead code, without loss of generality. A *strongly connected component* is a subgraph $H = (N', E')$ of a directed graph $G = (N, E)$ such that each $n \in N'$ is reachable from every $m \in N'$ by a path that uniquely consists of edges in E', where $N' \subseteq N$ and $E' \subseteq E$. A *directed acyclic graph* is a digraph that contains no directed cycles.

In a directed graph G a node x *pre-dominates* (*post-dominates*) a node y if every directed path from $start$ to y (y to $exit$) passes through x. A nodes x *dominates* a node y if x pre-dominates y or x post-dominates y. The *pre-dominator* and *post-dominator* trees of a graph $G = (N, E)$ represent the pre-dominance and post-dominance relation, respectively, between N. The parent x of y in the pre-dominator tree is its immediate pre-dominator, denoted $ipredom(y) = x$; conversely, the parent y of x in the post-dominator tree is its immediate post-dominator, denoted $ipostdom(x) = y$. The *least common ancestor* (LCA) of a pair of nodes x and y in a tree T is the ancestor of u and v in T that is located farthest from the root. The LCA l of a set of nodes $S \in T$ is the LCA of all unordered pairs of S, denoted $lca(S) = l$.

3 Taxonomy of Instrumentation

A fundamental issue when attempting to combine MB data through static techniques is the placement of ipoints with respect to the program under consideration. Evidently, coarse-grained tracing mechanisms require greater support from test-data generation methods, which consequently require assistance from coverage metrics, both of which remain open problems within the scope of timing analysis. We propose a taxonomy of available tracing mechanisms (software, compiler-generated, hardware, and CPU simulator) with the aim of later classifying their effect on our analysis. Following are *instrumentation profiles.*

1. *Full*: An ipoint is placed at the beginning of each instruction, which is easily achieved when using CPU simulators.
2. *Block*: An ipoint is placed at the beginning, the end, or both the beginning and end, of each basic block.
3. *Optimal*: Optimal tracing mechanisms have been introduced [4], which enable the entire traversed path through a program to be reconstructed from a trace file with the minimum number of ipoints.
4. *Coverage*: Code coverage analysis techniques often place software probes at the beginning of basic blocks that exhibit certain properties with respect to the dominance relation. For example, Agrawal [1] utilises the dominator graph data structure to find a minimal instrumentation placement with respect to block and branch coverage metrics.
5. *Branch*: Nexus [15] is a hardware debug interface that collects tracing data at program flow discontinuities, i.e. at taken conditional branches and exceptions.
6. *Structural*: The CFG is (partially) instrumented such that certain structural properties of the resultant IPG hold, which are often necessitated to facilitate algorithms operating on the IPG, e.g. reducibility. We further elaborate on this profile during the course of the paper.
7. *Arbitrary*: The number and placement of ipoints are not restricted.

This taxonomy is important because it determines the precision and resolution of further analysis. The full and block profiles enables the WCET of each BB to be extracted, thus it inheres with a myriad of static analysis techniques already outlined in the literature, which are based on BBs. The optimal, coverage, and branch profiles normally constrain the subsequent calculation to techniques that can handle irreducible graphs. The structural profile is principally used when tree-based calculations are required, since trees typically experience difficulties handling irreducible regions. It is difficult to qualify the impact of an arbitrary profile, since it might equate to another profile. However, in the worst case, a program will only record time stamps at the beginning and end of execution, which corresponds to current industrial practice of black box testing. We stress that how to instrument a program for timing analysis purposes remains an open question and is not the focus of this paper.

Note that ipoints do not have to correspond to extra code physically positioned in the program. They can correspond to notional points in the program where an observation takes place: either intrusively by calls to a tracing library, or completely transparently by using hardware tracing mechanisms or CPU simulators. Furthermore, in every profile, we assume that $start$ and end are ipoints of each function so that the contribution of each function to the final WCET can be gauged.

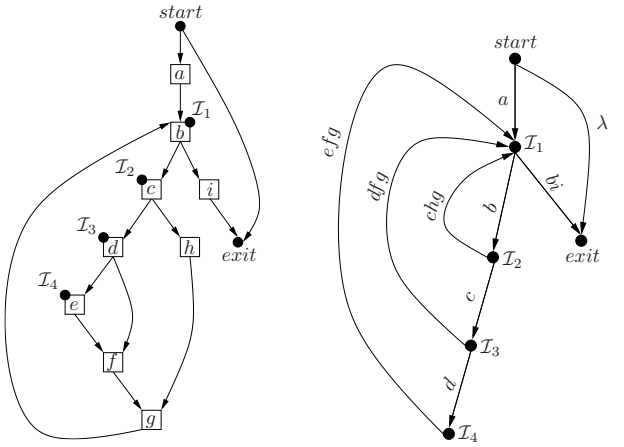
4 Instrumentation Point Graphs

The underlying data structure of our measurement-based approach is the instrumentation point graph (IPG), which

arranges the transitions among ipoint pairs in the CFG into structural form on an intra-procedural basis. The first step in constructing the IPG is to re-structure the CFG into a CFG* representation, which is a canonical form of the CFG in which there are two types of nodes: the set $\hat{\mathcal{I}}$ of ipoints represent individual time stamp instructions; the set $\hat{\mathcal{B}}$ of *basic sub-blocks* represent sequences of functional program instructions[2]. The definition of an IPG follows naturally from this construction:

Definition 1. *An **IPG** is a connected, directed graph* $\Gamma = (\hat{\mathcal{I}}, E', s, e)$ *such that* $(i, j) \in E'$ *if and only if there exists a path* $i \rightarrow b_1 \rightarrow \ldots \rightarrow b_n \rightarrow j$ *such that each* $b_i \in \hat{\mathcal{B}}$ *and* $n \geq 0$.

Figure 1(a) shows a CFG that has been augmented with a set of ipoints $\{\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4\}$, which have been placed at the beginning of BBs $\{b, c, d, e\}$. The resultant IPG generated from these ipoints is shown in figure 1(b). Each edge has been labelled with sequences of BBs executed between pairs of ipoints in the CFG, which we describe next; instructions thus reside on edges, whereas they reside in BBs in the CFG.



(a) CFG probed with ipoints

(b) IPG generated from structural profile

Figure 1. Example CFG and IPG

Instruction Blocks

A key aim in a hybrid MB WCET framework is the reduction of the *probe effect*, in which ipoints collecting timing data affect the execution time of a program in an adverse manner. A consequence of the reduction in ipoints is the introduction of a new unit of computation, for calculations based on the IPG, to the time observed between ipoints, which we term *instruction blocks* (IB). An IB contains a subgraph $H = (N', E')$ of the CFG* that is traversed between pairs of ipoints, although this might be intractable to compute in an arbitrary instrumentation profile since arbitrary subgraphs might reside between pairs of ipoints. Significantly, instructions of a unique BB can reside in several IBs, thus it is not possible to extract the WCET of individual BBs. This is exhibited in the IPG of figure 1(b): $\mathcal{I}_2 \rightarrow \mathcal{I}_1$, $\mathcal{I}_3 \rightarrow \mathcal{I}_1$, and $\mathcal{I}_4 \rightarrow \mathcal{I}_1$ all contain context-sensitive execution of BB g. However, this context-sensitivity also permits tree-based calculations to become more path-aware since the WCET of BBs are no longer considered in isolation, thus resulting in more accurate WCET estimates given sufficient testing and coverage. Note that the transition $start \rightarrow exit$ contains λ, which signifies that no functional instructions are executed whenever this edge is followed.

[2]In the arbitrary instrumentation profile, software probes can reside in any location inside a BB, thus a number of sub-nodes are spawned

Iteration Edges and Reducibility

In control-flow analysis and global code optimisation techniques, it is important to distinguish loops in the CFG. This can be achieved by determining loop backedges $u \rightarrow v$ such that v pre-dominates u and that v pre-dominates all nodes contained in the loop body. Furthermore, the loop-nesting level of a loop indicates the containment relationship of that loop amongst a set of (nested) loops. However, the straightforward loop identication process is inhibited by mutliple-entry loops where a set of nodes collectively pre-dominate the nodes in the loop [19]: such loops are termed *irreducible*. In these cases it is often difficult to compute the nodes that are contained in an irreducible loop body, the nesting relationship among loops, and even the number of loops.

Loop backedges of the IPG are a subset of a more general class that we term *iteration edges*. We shall later observe that iteration edges permit the integration of more path-based information into tree-based calculations, especially when the CFG has been sparsely instrumented. An iteration edge $u \rightarrow v$ is one in which u, v belong to the same loop-nesting level and includes the traversal of a CFG* loop backedge in its instruction block. However, identification of iteration edges is not straightforward unless the IPG is reducible, which cannot be assumed to hold without careful selection of ipoint positions with respect to the CFG. Although techniques do exist that identify irreducible loops [10, 19], there still remains incoherent definitions about what constitutes a loop in an arbitrary graph [18]. Clearly, misidentifying such edges can lead to the possibility of underestimation in the calculation stage, since they should be factored by an appropriate loop bound.

In essence, we want to instrument a CFG with a set of loops $L = \{L_1, \ldots, L_n\}$ and be able to identify a set of IPG loops $\mathcal{L} = \{L'_1, \ldots, L'_m\}$, $m \geq n$, such that:

1. There is a function $f : L_i \rightarrow \hat{\mathcal{L}}$, where $\hat{\mathcal{L}} = \{L'_1, \ldots, L'_k\}$ with $1 \leq k \leq m$,
2. Each $L'_j \in \hat{\mathcal{L}}$ is reducible and shares the same header.

It is easy to show that the first condition holds by placing a non-empty set of ipoints into each loop L_i, because there is subsequently a strongly connected component of these ipoints in the IPG. For an arbitrary CFG, a baseline optimal instrumentation profile always satisfies this property due to the fact that the entire traversed path can be re-constructed from a trace file, i.e. each iteration of each loop in the CFG must be observed. The second condition is only satisfied by ensuring a unique ipoint in each L_i pre-dominates all other ipoints in L_i, which requires examination of the pre-dominance relation between nodes and edges of L_i, depending on where ipoints are to be placed. This is stated more precisely in the following theorem.

Theorem 1. *Let* L *be a loop with a set of loop headers* $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ *in the CFG and* $\hat{\mathcal{L}}$ *the corresponding loops in the IPG. Each* $L'_j \in \hat{\mathcal{L}}$ *is reducible and shares the same header if and only if a non-empty subset* M *of* L *is*

instrumented such that there exists a unique $x \in M$ which pre-dominates all nodes in M.

Proof. Omitted, see [6] for details. □

This theorem is principally important since it permits a reducible CFG to be instrumented so that each set of loops in the resultant IPG can be distinguished using standard loop identification techniques. That is, each iteration edge is also a loop backedge, which is a property we exploit in the next section. The question of whether instrumentation profiles outlined above - particularly the optimal one - can be modified to comply with this theorem remains open. A further outcome of this result is the conjecture that some irreducible CFGs can be instrumented to ensure IPG reducibility, although this would require well-defined notions of a loop header and backedge in an irreducible CFG.

5 Tree-Based WCET Analysis

In the introduction we noted that our aim is to combine data acquired during measurement with high-level SA techniques in order to obtain a final WCET estimate. This issue is now addressed by describing in detail an algorithm that performs a hierarchical decomposition of the IPG into an *Itree*, the properties of which are equally specified.

The requirement for a new tree representation is motivated by the fact that current tree-based methods do not pertain to the IPG whenever coarse-grained instrumentation profiles are used. Only the full instrumentation profile results in basic blocks being the basic unit of computation; therefore, instruction blocks, which incorporate context-sensitive execution of basic blocks, are used instead. The tree representation that we present thus permits instruction blocks to be combined without undue pessimism. A further and more pertinent motivation arises from the problem of modelling iteration edges with respect to a unique CFG loop, since this regularly occurs using coarse-grained instrumentation profiles. For these cases, we introduce a new tree construct, whilst noting that the approach outlined by Petters *et al.* [17] has not addressed this fundamental issue.

During the description of the Itree representation and algorithm, we refer to each loop of the IPG as a tuple (h, T), in which h is the header that pre-dominates T, and each $t_i \in T$ is a tail with a backedge to h. Moreover, we distinguish between: *for/while* loops in which h is the immediate post-dominator of each $t_i \in T$, henceforth referred to as a *for* loop; *do-while* loops in which no $t_i \in T$ post-dominates h, unless there is a unique $t \in T$, in which case t post-dominates h.

5.1 Itree representation

The Itree representation that we propose has similar properties to those of an abstract syntax tree since it models three high-level constructs: sequence, selection, and iteration. In addition, we introduce a novel construct, the *meta-loop*, which enables (sub-)paths through the CFG loop to be combined in the calculation stage, especially when it is possible to show that no unique worst path is followed on each iteration. In this pathological case, estimates are equally as safe since the meta-loop structure enables the worst path to be determined.

The *leaves* of an Itree correspond to a transition among a pair of inodes $(\mathcal{I}_i, \mathcal{I}_j)$ since the basic unit of computation is the instruction block. There are four kinds of interior nodes, all of which are exemplified in figure 4:

- An *alternative* node is a rooted n-ary tree; it either models a selection of paths $\mathcal{I}_i \xrightarrow{+} \mathcal{I}_j$ from a branch node $\mathcal{I}_i$ to $\mathcal{I}_j = ipostdom(\mathcal{I}_i)$; or, it models a selection of paths $T \xrightarrow{+} lca(T)$ in a *do-while* loop construct, where T is a set of loop tails such that $|T| > 1$.
- A *sequence* node is a rooted n-ary tree, which models the path $\mathcal{I}_b \rightarrow \mathcal{I}_s \xrightarrow{+} \mathcal{I}_p$, in which $\mathcal{I}_b$ is a branch node or a loop header with an immediate successor $\mathcal{I}_s$, and there is a non-empty path from $\mathcal{I}_s$ to $\mathcal{I}_p = ipostdom(\mathcal{I}_b)$.
- A *loop* node is a rooted binary tree that models the paths in a loop with header h and tail t. The right tree is a leaf representing the loop iteration edge $t \rightarrow h$. The left tree either represents the path $h \xrightarrow{+} t$ when h is the destination of a unique tail t, or it represents the path $b \xrightarrow{+} t$ in which $t \in T$, $|T| > 1$, and b is the nearest branch node of t, which shall be explained shortly. In the case of a self-loop, the left tree is always empty.
- A *meta-loop* is a rooted n-ary tree that models several sub-loops with tails $\{t_1, \ldots, t_n\}$ such that each t_i has the same header h and shares a path $p : h \xrightarrow{*} b$ in which b is the nearest branch node to each t_i: p is modelled as the $n + 1^{th}$ subtree of the meta-loop construct.

5.2 Itree construction algorithm

The construction of the Itree assumes that a CFG has been instrumented using the result of theorem 1 so that each loop in the resultant IPG $\Gamma = (N', E', start, exit)$ is reducible. There are several further assumptions that contribute to the notion of the structural profile, which are required for correctness. These all pertain to the structural properties of each IPG loop L, and are summarised as follows:

1. Each backedge $t_i \in T \rightarrow h$ traverses the *same, unique* CFG* backedge in its respective instruction block,
2. Each $t_i \in T$ is the source of a unique backedge,
3. Every $n \in L - T - \{h\}$ is either post-dominated by h (whenever L is a *for* loop) or collectively post-dominated by some subset of T (whenever L is a *do-while* loop),
4. Any branch node b on a path $h \xrightarrow{+} t_i$, which is *not* post-dominated by t_i, pre-dominates t_i.

We motivate the first two assumptions by the fact that it is often difficult to determine loop nesting levels whenever a header is the destination of backedges that correspond to different loops, or whenever tails have multiple edges to different headers. Furthermore, these allow the subsequent calculation to safely combine sub-loops contributing to a unique CFG loop. The third assumption enables us to determine that every exit from L must be from h in *for* loops, or from one of $t_i \in T$ in *do-while* loops. The fourth assumption is required in the construction of meta-loop subtrees due to an auxiliary data structure that we utilise, the *compressed pre-dominator tree*, which is explained shortly. We envisage that some of these assumptions will

be relaxed in order to handle a more general class of IPG, which is currently under investigation.

There are several preprocessing steps to the main algorithm. First, the pre-dominator and post-dominator trees of Γ are built using standard dominator tree algorithms [3]. Second, for each loop L in Γ, ordered (h, T) pairs are computed, which can be achieved by identifying all backedges $u \rightarrow v$ in which v pre-dominates u as we assume that all loops are reducible. During the backedge identification process, Γ is transformed into a directed acyclic graph Γ', such that all backedges identified are removed. The construction of the Itree requires that $start \rightarrow exit \in E'$ to prevent premature and incorrect termination of the algorithm. When $start$ has a unique successor, namely $exit$, the Itree consists of a single instruction block $start \rightarrow exit$; otherwise, Itree construction is initiated by calling algorithm 1 with parameters $source = start$ and $target = exit$, noting that the entire algorithm operates on Γ'. There are several sub-algorithms (c.f. algorithms 1, 2, 3, 4, 5) that contribute to the overall construction of the Itree, thus we give individual focus to each in turn. Figure 4 shows the Itree constructed from the IPG in figure 1(b).

```
BuildALTRoot (branch, target)
root = ALT
foreach s_i ∈ succ(branch) do
    if s_i == target then
        root.i^th Tree = branch → s_i
    else
        seqRoot = SEQ; seqRoot.1^st Tree = branch → s_i
        root.i^th Tree = BuildSEQRoot (seqRoot, s_i, target)
return root
```

Algorithm 1: Build alternative subtree

```
BuildLOOPRoot (header, tail)
root = LOOP; root.rightTree = tail → header
if tail ≠ header then
    if header post-dominates tail then
        header' = ComputeDummyNode (header, tail)
    else
        header' = header
    root.leftTree = WhichSubTree (header', tail)
return root
```

Algorithm 2: Build loop subtree

Alternative subtree (Algorithm 1)

This algorithm builds an alternative subtree for a branch node b with an immediate post-dominator p. For each immediate successor s_i of b, the algorithm first resolves whether s_i is p, in which case $b \rightarrow s_i$ is inserted as the i^{th} tree of the alternative root. If not, this implies that there is a sequence of sub-paths from s_i to p, thus a sequence subtree is constructed as the i^{th} tree: its first tree is $branch \rightarrow s_i$, and all remaining subtrees to p are built through a call to algorithm 5.

Loop subtree (Algorithm 2)

This algorithm builds a loop subtree for a loop L with a header h and a unique tail t. In building the loop body subtree, we first check that h is not a self-loop. When this is not the case, the algorithm determines whether the loop is a *for* or *do-while* structure by utilising the post-dominance relation amongst the header and the tail outlined above.

In the case of a *for* loop, the problem occurs that a subset of $succ(h)$ lie *outside* the loop body since h is post-dominated by some $y \notin L$; therefore, a dummy node h' is created in which each $s' \in succ(h')$ resides in L, noting that each $s \in succ(h)$ that belongs to L must be post-dominated by t. The loop body is thus constructed with a call to algorithm 4 in which h' is the source and t the target. On the other hand, *do-while* structures do not experience this problem since we assume that each node in L is post-dominated either by h or by t; instead, h must be flagged as a non-header to avoid infinite recursion before calling algorithm 4 with h as the source and t the destination.

```
BuildMETARoot (header, T)
currentHeader = header
C = ComputeCompressedDominatorTree (header, T)
for level_C = height(C) downto 0 by −1 do
    foreach x ∈ C with level(x) == level_C do
        if x ∈ T then
            root = LOOP
            root.rightTree = n → currentHeader
            retrieve dummy node d of parent(x)
            if x ≠ currentHeader then
                root.leftTree = WhichSubTree (d, x)
            tailRoots(x) = root
        if x source of multiple compression edges then
            root = META; k = 1
            foreach y ∈ C with best(y) == x do
                if y ∈ T then
                    root.k^th Tree = tailRoots(y); k = k + 1
                if y source of multiple compression edges then
                    root.k^th Tree = metaRoots(y); k = k + 1
            if x ≠ currentHeader then
                retrieve dummy node d of parent(x)
                root.k^th Tree = WhichSubTree (d, x)
            metaRoots(x) = root
root = metaRoots(currentHeader)
return root
```

Algorithm 3: Build meta-loop subtree

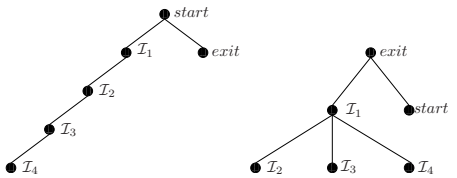
Meta-loop subtree (Algorithm 3)

This algorithm builds a meta-loop subtree for a loop L with a header h and a set of loop tails T; $currentHeader$ is set to h, which is useful in algorithm 5. This algorithm depends on the construction of an auxiliary data structure, the *compressed pre-dominator tree* (CPT), which can be constructed (off-line) once all (h, T) tuples have been computed. In essence, we want to construct the body of each sub-loop for t_i from the nearest (predecessor) branch node b that t_i does not post-dominate; b is either another loop tail t_j or the node at which control flow diverges towards t_i in L. Therefore, the path $p : h \xrightarrow{*} b$ should not be included in this body since p can be followed whenever the body of some other tail t_j is executed.

The CPT thus provides a convenient way of extracting the nearest branch b for each tail, and the nearest branch b' for each b. It is iteratively computed through a set of nodes Q such that, initially, $Q = T$. For each $q \in Q$, a backwards traversal of the pre-dominator tree is performed until a node q' is found that q does not post-dominate; q' then becomes the parent of q in the CPT. Whenever q' is not a member of the CPT, it is added to a set Q', and at the end of each iteration $Q = Q'$. This process continues until Q contains a unique member q; if q is *not* the header h, then h becomes the parent of q since each t_i clearly shares the path $h \xrightarrow{+} q$. In a further step, we compress the paths of each q in the CPT, in which $parent(q)$ is a tail with a *unique* child, to its

first ancestor a that either has multiple children or is not a loop tail. The result is stored in $best(q)$, whereby $best(q)$ is initially q. This is because the subtree of each q with this property should be a subtree of the meta-loop rooted at a, and not the subtree rooted at $parent(q)$. This is easily achieved by a depth-first search on the CPT.

As an example of the construction of the CPT, we return to the IPG of figure 1(b) in which the pre-dominator and post-dominator trees are shown in figures 2(a) and 2(b), respectively. Initially, $Q = \{\mathcal{I}_2, \mathcal{I}_3, \mathcal{I}_4\}$. Following the procedure outlined above, at the end of the first iteration, we have $parent(\mathcal{I}_2) = \mathcal{I}_1, parent(\mathcal{I}_3) = \mathcal{I}_2, parent(\mathcal{I}_4) = \mathcal{I}_3$, and $Q' = \{\mathcal{I}_1\}$, thus resulting in termination as $Q = Q'$ contains a unique element. We now have to compress paths since the parents of $\mathcal{I}_3$ and $\mathcal{I}_4$ are both tails with a unique child in the CPT. Performing a depth-first search on the CPT, $best(\mathcal{I}_3) = parent(\mathcal{I}_2)$ since $best(\mathcal{I}_2) == \mathcal{I}_2$, and $best(\mathcal{I}_4) = best(\mathcal{I}_3)$ since $best(\mathcal{I}_3) \neq \mathcal{I}_3$. This results in the CPT shown in figure 3 such that solid edges are tree edges and dashed edges are path-compression edges.



(a) Pre-dominator Tree (b) Post-dominator Tree

Figure 2. Dominator trees of IPG in figure 1(b)

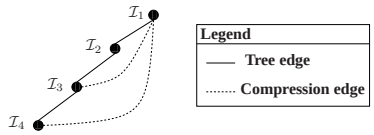


Figure 3. Compressed pre-dominator tree

The meta-loop construct is subsequently built by visiting the nodes of the CPT level by level in a bottom-up fashion. Let X denote the nodes at level i: two properties of each $x \in X$ are examined. First, whenever x is a loop tail, a loop subtree is built in which the loop body contains the subgraph from the nearest branch of x, i.e. $parent(x)$ in the CPT, to x. The loop subtree constructed is stored in an array $tailRoots$ for subsequent retrieval, which is indexed by x.

Second, whenever x has multiple children in the CPT, x is a branch node in Γ' such that a subset of tails T' share the path $p : h \xrightarrow{*} x$[3]. Therefore, we want to build a meta-loop subtree in which the loop subtree of each $t' \in T'$ sharing p is a child. The loop subtree of each t' with $best(t') == x$ is thus retrieved by indexing $tailRoots$ with t', noting that the bottom-up traversal of the CPT guarantees that the loop subtree of t' has already been constructed. Finally, p is built for the meta-loop construct of x by retrieving $parent(x)$ in the CPT; it is stored in an array $metaRoots$ for analogous reasons to $tailRoots$, as the meta-loop construct of x might itself be a descendant of another meta-loop subtree that is subsequently constructed.

Each time $parent(x)$ is fetched, a *dummy* node d must be created before building the subtree $parent(x) \xrightarrow{+} x$, which is necessitated for one of two reasons. On the one hand, $parent(x)$ has several children in the CPT, in which case there are successors of $parent(x)$ in Γ' that do not converge on x. On the other hand, all successors of $parent(x)$ in Γ' do converge on x, *but* the immediate post-dominator of $parent(x)$ is not a node post-dominated by x, thus the algorithm will never reach x. For example, in figure 2(b), we see that $ipostdom(\mathcal{I}_2) == \mathcal{I}_1$, and that $parent(\mathcal{I}_3) == \mathcal{I}_2$, thus the loop body of $\mathcal{I}_3$ will not be correctly constructed. The dummy node d overcomes both of these problems: a successor s of $parent(x)$ in Γ' is inserted into $succ(d)$ if and only if s is post-dominated by x; furthermore, the immediate post-dominator of d is computed by taking the least common ancestor of $succ(d)$ whenever $|succ(d)| > 1$, or the unique $s \in succ(d)$; thus, construction of the subtree $d \xrightarrow{+} x$ must always converge on x. For example, the dummy node d created for the loop body of $\mathcal{I}_3$ has successors $\{\mathcal{I}_3\}$ and since $|succ(d)| == 1$, $ipostdom(d) = \mathcal{I}_3$.

```
WhichSubTree (d, target)
if d is tail then
    seqRoot = SEQ
    seqRoot .1st Tree = body: (currentHeader → d)
    return BuildSEQRoot (seqRoot, d, target)
else if ipostdom(d) ≠ target then
    seqRoot = SEQ
    if |succ(d)| > 1 then
        seqRoot .1st Tree = BuildALTRoot (d, ipostdom(d))
    else
        seqRoot .1st Tree = d → ipostdom(d)
    return BuildSEQRoot (seqRoot, ipostdom(d), target)
else
    if |succ(d)| > 1 then
        return BuildALTRoot (d, target)
    else
        return d → target
```

Algorithm 4: Determine subtree construction call

Determine subtree construction (Algorithm 4)

This is a helper procedure used in combination with algorithms 2 and 3. Specifically, it determines the type of subtree to be built each time a dummy node d has been created.

When d is identified as a tail, this indicates that some subtree in a meta-loop construct shares the loop body of d. We thus want to avoid the quadratic time complexity that would arise by rebuilding the loop body $currentHeader \xrightarrow{+} d$ for each node that is a descendant of d in the CPT. For example, in figure 3, we see that $\mathcal{I}_3$ shares the loop body of $\mathcal{I}_2$ and that $\mathcal{I}_4$ shares the loop body of $\mathcal{I}_3$ (and implicitly that of $\mathcal{I}_2$), thus we want to avoid re-building these loop bodies. This is achieved by inserting the first subtree of the sequence root as a reference to the loop body that it shares, which is $currentHeader \rightarrow d$. The construction of the sequence root is completed through a call to algorithm 5.

Otherwise, the complete path $p : d \xrightarrow{+} target$ needs to be built. When the immediate post-dominator of d is *not* $target$, this means that the concatenation of paths $q : d \xrightarrow{+} ipostdom(d)$ and $r : ipostdom(d) \xrightarrow{+} target$ form p. That is, a sequence subtree needs to built in which its first subtree models q, and all subsequent subtrees model r, as r itself might be composed of several sub-paths. To construct the first subtree, the number of successors of d are examined. A branch node invokes a call to algorithm 1; otherwise, the transition $d \rightarrow ipostdom(d)$ is inserted, since $ipostdom(d)$

[3] p is empty whenever $x == h$ and non-empty otherwise

must be the unique successor of d. Analogous to when d is a loop tail, a call to algorithm 5 completes the sequence subtree.

When the immediate post-dominator of d is *not* $target$, this indicates that $d \xrightarrow{+} ipostdom(d)$ and p are equivalent, thus a sequence subtree is not required. If d is a branch node, an alternative subtree is returned, else the transition $d \rightarrow target$.

```
BuildSEQRoot (seqRoot, source, target)
y = source; k = 2
while y ≠ target do
    if y is header then
        T = tails(y)
        if |T| > 1 then
            seqRoot.k^th Tree = BuildMETARoot (y, T)
        else
            seqRoot.k^th Tree = BuildLOOPRoot (y, t ∈ T)
        k = k + 1
        if y post-dominates T then
            y = ComputeNonHeader (y)
        else
            // Details omitted
            seqRoot.k^th Tree = BuildDoWhileExitRoot (T)
            y = lca(T) in post-dominator tree; k = k + 1
            if y == target then
                return seqRoot
    else
        z = ipostdom(y)
        if |succ(y)| > 1 then
            seqRoot.k^th Tree = BuildALTRoot (y, z)
        else
            seqRoot.k^th Tree = y → z
    y = z; k = k + 1
return seqRoot
```

Algorithm 5: Build sequence subtree

Sequence subtree (Algorithm 5)

This algorithm completes the construction of a sequence subtree, which is called from algorithms 1 and 4. Since the first tree of the sequence subtree is already constructed before calling this algorithm, the subtree counter k is set to two; these subtrees are then iteratively built from node y until $target$ is reached. On each iteration, the algorithm determines which property is pertinent to y.

If y is a loop header, a meta-loop or loop subtree is constructed as the k^{th} tree, depending on the number of tails with destination header h, as described above. Next, we determine from where the loop exits: in the case of a *for* loop, it is from h as we assume that all nodes in L are post-dominated by h; in the case of a *do-while* loop, it is from the set of loop tails T as we assume that all nodes in L are post-dominated by some $t \in T$. We require support from another algorithm to build the exit subtree of a *do-while* loop with multiple tails, but we have omitted the details due to space restrictions; however, it adopts similar principles to that of the meta-loop construction in the use of a compressed *post-dominator* tree, see [6] for details. If y is not a loop header, the immediate post-dominator of y is retrieved and stored in z. Either y is a branch node, in which case algorithm 1 is called with $source = y$ and $target = z$; or, y has a unique successor and thus the transition $y \rightarrow z$ is inserted. At the end of each iteration, y is updated with the value of z.

5.3 Calculations on Itrees

The timing schema rules to compute WCET estimates on an Itree are the same as those introduced by Park and

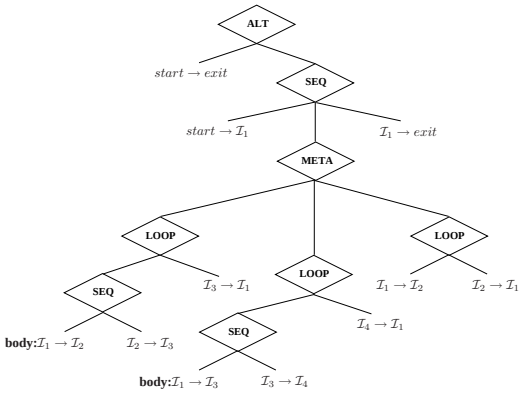


Figure 4. Itree constructed from figure 1(b)

Shaw [16] in the cases of sequence and alternative, except that the units of computation are instruction blocks, and not basic blocks. Following are the rules for loop and meta-loop structures.

$$WCET(loop) = (WCET(body) + WCET(iteration\ edge)) * k \quad (1)$$

$$WCET(meta) = \sum_{i=1}^{n} WCET(loop_i) + WCET(p) * \sum_{i=1}^{n} k_i \quad (2)$$

In equation (1), k denotes the maximum *observed* bound for each iteration edge, depending on the use of suitable coverage criteria. It is possible to integrate information of loop bounds derived from a suitable static analysis technique to prevent underestimation, the details of which are considered beyond the scope of the paper. Note that the WCET of an IPG loop includes the sum of its body *and* the iteration edge since instructions reside on edges in the IPG. In equation (2), the WCET of a meta-loop is the sum of all sub-loops plus the WCET of any path p common to all sub-loops. Since p is always executed when any sub-loop is exercised, it can be factored by the sum of the number of iterations of all sub-loops. The value k_i denotes the loop bound of each $loop_i$ that is extracted during measurement. As IPGs are generated on an intra-procedural basis, a bottom-up traversal of a call graph is required to derive WCET estimates for multi-procedural programs. Note that empty Itrees are not permissible as we assume that each CFG has inodes $start$ and $exit$.

6 Evaluation

In this section we demonstrate how tree-based calculations centred on IPGs become more path-aware and how the meta-loop structure can contribute to tighter WCET estimates given sufficient coverage. The motivation of this paper has centred around partially instrumented programs, and *not* on a comparison between SA and MB techniques.

Reconsider the CFG of 1(a) and the IPG of 1(b): table 6 shows the WCET of each basic block (BB) and instruction block (IB), assuming these have been computed through simulation or some MB technique. Note that the WCET of an IB $\hat{\mathcal{I}}$ can never be greater than the sum of the WCETs of each BB that is a member of $\hat{\mathcal{I}}$ *if* the WCET of BBs have

been extracted from a SA technique, assuming the absence of destructive interference that may occur through timing anomalies [13].

BB	WCET(BB)	IB	WCET(IB)
a	10	$start \rightarrow exit$	0
b	15	$start \rightarrow \mathcal{I}_1$	10
c	12	$\mathcal{I}_1 \rightarrow exit$	20
d	8	$\mathcal{I}_1 \rightarrow \mathcal{I}_2$	15
e	20	$\mathcal{I}_2 \rightarrow \mathcal{I}_1$	40
f	16	$\mathcal{I}_2 \rightarrow \mathcal{I}_3$	12
g	10	$\mathcal{I}_3 \rightarrow \mathcal{I}_1$	32
h	25	$\mathcal{I}_3 \rightarrow \mathcal{I}_4$	8
i	6	$\mathcal{I}_4 \rightarrow \mathcal{I}_1$	40

Figure 5. WCET of units of computation of figures 1(a) and 1(b)

Let us assume that the maximum number of iterations of the CFG loop consisting of backedge $g \rightarrow b$ is 50. A simple tree-based approach would assume that the worst path in the CFG loop is followed on each iteration, which is $b+c+MAX(h, d+e+f)+g = 15+12+MAX(25, 8+20+16)+10 = 81$. This value is then factored by the loop bound and subsequently added to the WCET of BBs a and i since these are always executed; thus, the WCET of the program is $(50 * 81) + 10 + 6 = 4066$. The approach presented in this paper also permits the selection of the longest path through the CFG loop on each iteration: this is clearly the loop consisting of iteration edge $\mathcal{I}_4 \rightarrow \mathcal{I}_1$, which equates to the path $bcdefg$. The WCET of this loop is $((15 + 12 + 8 + 40) * 50) = 3750$, and the final WCET estimate - combining transitions $start \rightarrow \mathcal{I}_1$ and $\mathcal{I}_1 \rightarrow exit$ with this value - is $3750 + 10 + 20 = 3780$. An improvement is in evidence (despite and because of partial instrumentation) since the worst path through the loop contains context-sensitive execution of the worst path for each BBs e, f, and g. Thus, it is the underlying unit of computation, the instruction block, that permits our tree-based calculations to become more path-aware.

However, our approach also permits more path-based information pertaining to loops to be integrated into the calculation stage using the meta-loop structure. For example, let us now assume that during testing we have ascertained that each IPG loop has the following maximum number of iterations: loop with backedge $\mathcal{I}_2 \rightarrow \mathcal{I}_1$, 15 iterations; loop with backedge $\mathcal{I}_3 \rightarrow \mathcal{I}_1$, 20 iterations; loop with backedge $\mathcal{I}_4 \rightarrow \mathcal{I}_1$, 15 iterations. In this case, the WCET of the meta-loop as depicted in figure 4 is $((15 + 40) * 15) + ((15 + 12 + 32) * 20) + ((15 + 12 + 8 + 40) * 15) = 3130$, and the final WCET estimate is $3130 + 10 + 20 = 3160$. Nevertheless, we emphasise that the applicability of this latter calculation depends on suitable testing and coverage, both of which remain open questions.

7 Conclusions and Future Work

In this paper, we have shown how to combine low-level measurement-based data through a tree-based approach on the instrumentation point graph (IPG). For these purposes, we have meticulously described an algorithm that decomposes an IPG into a hierarchical form that supports a novel structure, the meta-loop, which is able to combine several IPG sub-loops that collectively contribute to the WCET of a unique CFG loop. We presented the timing schema for the Itree, and have also shown how the meta-loop structure enables tight estimations but also how to err on the side of caution in the absence of sufficient coverage.

The focus of our future work surrounds coverage criteria for the various instrumentation profiles, especially the optimal profile as this permits the longest path to be derived. Moreover, many instrumentation profiles lead to irreducible IPGs, thus we are currently investigating a technique that is able to handle these cases, since tree-based methods fail. For this goal, we want to reshape path-based and IPET calculation methods so that they pertain to the IPG.

References

[1] H. Agrawal, "Dominators, Super Blocks, and Program Coverage", *In proceedings of the* 21^{st} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, Jan. 1994.

[2] A.Aho, R.Sethi, and J. Ullman, "Compilers: Principle, Techniques and Tools", Addison-Wesley, 1986.

[3] S. Alstrup, D. Harel, P.W. Lauridsen, and M. Thorup, "Dominators in Linear Time", *In SIAM Journal of Computing*, 28(6): 2117-2132, Dec. 1999.

[4] T. Ball and J.R. Larus, "Optimally Profiling and Tracing Programs", *In Proceedings of the* 19^{th} *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, Feb. 1992.

[5] I. Bate and R.Reutemann, "WCET Analysis for Dynamic Branch Predictors", *In Proceedings of the* 16^{th} *Euromicro Conference of Real-Time Systems (ECRTS'04)*, July 2004.

[6] A. Betts and G. Bernat, "Tree-Based WCET Analysis on Instrumentation Point Graphs", Technical Report, University of York, York, Oct. 2005.

[7] A. Colin and I. Puaut, "A Modular & Retargetable Framework for Tree-based WCET Analysis", *In Proceedings of the* 13^{th} *Euromicro Conference of Real-Time Systems (ECRTS'01)*, July 2001.

[8] A. Colin and S. Petters, "Experimental Evaluation of Code Properties for WCET Analysis", *In Proceedings of the* 24^{th} *Real-Time Systems Symposium (RTSS'03)*, Dec. 2003.

[9] G. Frantz, "Digital Signal Processor Trends", *IEEE Micro*, vol. 20, no. 6, pages 52-59, Nov. 2000.

[10] P. Havlak, "Nesting of Reducible and Irreducible Loops", *In ACM transactions on Programming Languages and Systems*, vol. 19, no. 4, pages 557-567, July 1997.

[11] Y-T.S. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", *In Proceedings of the ACM SIGPLAN Worskshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'95)*, Nov. 1995.

[12] X. Li, A. Roychoudhury and T. Mitra, "Modeling Out-of-Order Processors for Software Timing Analysis", *In Proceedings of the* 25^{th} *Real-Time Systems Symposium (RTSS'04)*, Dec. 2004.

[13] T. Lundqvist and P. Stenstrom, "An intergrated path and timing analysis method based on cycle-level symbolic execution", *In Proceedings of the* 20^{th} *Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.

[14] F. Mueller, "Timing Analysis for Instruction Caches", *Real-Time Systems*, vol. 18, no. 2-3, pages 217-247, May 2000.

[15] The Nexus 5001 forum. At http://www.nexus5001.org, Oct. 2005.

[16] C.Y. Park and A.C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema", *In IEEE Computer*, 24(5):48-57, May 1991.

[17] S.M. Petters, A. Betts, and G. Bernat, "A New Timing Schema for WCET Analysis", *In Proceedings of* 4^{th} *International Workshop on Worst Case Execution Time Analysis*, June 2004.

[18] G. Ramalingam, "On Loops, Dominators, and Dominance Frontiers", *In ACM transactions on Programming Languages and Systems*, vol. 24, no. 5, pages 455-490, Sep. 2002.

[19] V.C. Sreedhar, G.R. Gao, and Y-F Lee, "Identifying Loops Using DJ Graphs", *In ACM transactions on Programming Languages and Systems*, vol. 18, no. 6, pages 649-658, Nov. 1996.

[20] F. Stappert, A. Ermedahl, and J. Engblom, "Efficient longest executable path search for programs with complex flows and pipeline effects", *In Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, Nov. 2001.

[21] J. Wegener and F. Mueller, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints", *In Proceedings of the* 4^{th} *Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[22] R.T. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon, "Timing Analysis for Data and Wrap-Around Fill Caches", *Real-Time Systems*, vol. 17, no. 2-3, pages 209-233, Nov. 1999.