Reducing the Size of the Constraint Model in Implicit Path Enumeration using Super Blocks

Adam Betts
Department of Computing
Imperial College London
London SW7 2AZ
Email: abetts@imperial.ac.uk

Abstract—Implicit Path Enumeration (IPE) is the most popular calculation technique used in WCET analysis as it generally provides the most precision. It utilises the Control Flow Graph (CFG) to build a constraint system consisting of, amongst others, structural constraints (derived from the CFG) and relative capacity constraints (loop counts relative to the innermost enclosing loop). However, solving the constraint system is an NP-hard problem and reducing the size of the model is therefore desirable.

This paper shows that, when the CFG loops are free of break-like constructs, the constraint system produced by IPE consists of many superfluous structural constraints in that their elimination does not detract from the precision of the WCET estimate. We demonstrate this fact by building an alternative constraint system from a new intermediate data structure called the Super Block Control Flow Graph (SB-CFG). Furthermore, we illustrate how the SB-CFG allows us to deduce when relative capacity constraints are actually needed — until now they have been considered mandatory to avoid overestimations with loops.

Comparing the two constraint systems across a large number of automatically generated control flow graphs, our results show there is: a 50% reduction in the number of variables; a 55% reduction in the number of constraints; and a 69% reduction in solving time.

I. INTRODUCTION

A real-time system consists of a set of co-operating tasks whose computations must complete before a specific deadline. Proving this property is met before the system goes live tacitly assumes that the Worst-Case Execution Time (WCET) of each task is available. However, determining the actual WCET is in general impossible and an estimated WCET has to suffice. Estimating the WCET is subject to two conflicting requirements. One the one hand the estimate must bound the actual value because real-time systems are often deployed in safety-critical settings where underestimation could lead to system failure. On the other hand, the (assumed positive) difference between the estimate and the actual value must be as close to zero as possible, otherwise the system could be incorrectly deemed unschedulable due to overestimation.

In general, the only way to deliver a safe approximation of the actual WCET is to compute an estimate through appropriate models of the system. The program model, e.g. the abstract syntax tree or Control Flow Graph (CFG), is pivotal to this analysis as it represents the set of all structurally feasible paths through the program, from which the calculation engine computes a WCET estimate.

Although various calculation engines have been proposed in the literature, Implicit Path Enumeration (IPE) [1], [2] is by far the most dominant. IPE uses the CFG to build up a constraint system — either an Integer Linear Program (ILP) or a constraint program — that is basically a variation of the circulation problem [3]. The variables of the constraint system are execution counts of basic blocks which, when assigned to by the solver, determine the contribution of each basic block to the overall WCET estimate. Structural constraints model how execution counts between basic blocks are related, while relative capacity constraints [2] bound loops relative to their innermost enclosing loops. Relations between execution counts of variables can be added for further precision, e.g. to eliminate infeasible paths in the WCET computation. Theoretically, however, ILPs and constraint programs are NP-hard problems [4] and more variables or constraints merely push the analysis towards this limiting property.

This paper shows that, when the CFG’s loops are free of break-like constructs, the standard IPE constraint system consists of many superfluous structural constraints; that is, their elimination does not detract from the precision of the WCET estimate. We demonstrate this fact by building an alternative ILP from a new intermediate data structure called the Super Block Control Flow Graph (SB-CFG). The key property of the SB-CFG is that it naturally partitions basic blocks into sets, so-called super blocks [5], whose execution counts must be equal. By contrast, the standard ILP models this information implicitly through the flow conservation property of the circulation problem, with the penalty of extra variables and constraints.

Another benefit of the SB-CFG is that it allows us to formalise when relative capacity constraints are actually needed. Although Puschner and Schedl [2] gave an example of a CFG which required relative capacity constraints, they did not specify the general conditions under which their inclusion is mandatory in order to avoid overestimation. We show that simple analysis of the SB-CFG with respect to loop headers overcomes this limitation. Not only does this decrease the size of the model, it eases preceding analyses that gather bounds on loops: in cases where relative capacity constraints are not needed, it suffices to supply a global execution count on the loop header.

We have implemented the ILP variation of both constraint systems (the CFG and the SB-CFG) and applied them to a large number of automatically generated CFGs. Overall the results show that there is: a 50% reduction in the number of
variables; a 55% reduction in the number of constraints; and a 69% reduction in solving time. The practical benefit of our approach, therefore, is that it allows further constraints to be included in IPE, e.g., execution times on basic blocks, which actually affect the precision of the WCET estimate.

Following is a summary of our contributions:

1) Section IV: A presentation of the SB-CFG and how it is constructed from a CFG.
2) Section V: A mechanism to deduce when relative capacity constraints are needed in the constraint system using the SB-CFG.
3) Section VI: How to construct an ILP using the SB-CFG.
4) Section VII: Extensive evaluation of our ILP against the standard ILP on a large number of different CFGs.

The next two sections provide the necessary background to the remainder of the paper: Section II formalises the assumed program model while Section III scrutinises the standard ILP in more detail as it forms a central part of our analysis and evaluation.

II. PROGRAM MODEL

Before presenting the technical details of the program model, we need to review basic graph terminology and notation.

A. Graph Terminology and Notation

A graph \( G = (V_G, E_G) \) is a pair of finite sets \( V_G \) and \( E_G \), called vertices and edges, respectively, where \( E_G = \{(u, v) : u, v \in V_G\} \). We sometimes denote a directed edge \((u, v)\) as \( u \to v \) and say that \( u \) is the source and \( v \) is the destination. For any \( u \in V_G \): \( \text{pred}(u) = \{v : (v, u) \in E_G\} \) denotes the set of predecessors and \( \text{succ}(u) = \{v : (u, v) \in E_G\} \) denotes the set of successors. A path is a sequence \( v_1, v_2, \ldots, v_m \) such that, for all \( 1 \leq i < m \), \((v_i, v_{i+1}) \in E_G\).

A Strongly Connected Component (SCC) of a graph \( G \) is a maximal subgraph \( G' = (V_{G'}, E_{G'}) \) such that, for every \( u, v \in V_{G'} \), there are paths from \( u \) to \( v \) and vice versa. \( G \) is strongly connected whenever \( V_G = V_{G'} \); it is a Directed Acyclic Graph (DAG) if it does not contain SCCs composed of more than one vertex.

A tree \( T = (V_T, E_T, r) \) is a DAG in which \( r \in V_T \) is its root. For any \( v \in V_T \): \( v \) is a leaf if \( |\text{succ}(v)| = 0 \) or an internal vertex otherwise; its proper ancestors are the vertices on the unique path \( r \to v \), excluding \( v \); its descendants are the vertices that \( v \) has as an ancestor.

B. The CFG, Dominance, and the Loop-Nesting Tree

As stated above, IPE delivers a constraint system from a CFG, formally defined as follows:

Definition 1: A Control Flow Graph (CFG) \( C = (V_C, E_C) \cup \{t \to s\}, s, t \) is a directed graph such that:

- \( V_C \) is the set of basic blocks, which are maximal sequences of instructions consisting of a single entry point and a single exit point.
- \( s, t \in V_C \) are distinguished (dummy) vertices called, respectively, the entry vertex and the exit vertex. We assume that every basic block is reachable from \( s \) and can reach \( t \).
- \( E_C \) models the branches and fall-throughs between basic blocks. Every entry into the program has \( s \) as a predecessor and every exit out of the program has \( t \) as a successor.
- \( t \to s \in E_C \) to ensure that \( C \) is strongly connected.

Dominance is a key property in CFG analysis. For a CFG \( C \), a vertex \( u \) pre-dominates a vertex \( v \) if every path from \( s \) to \( v \) includes \( u \); similarly, a vertex \( u' \) post-dominates \( v \) if every path from \( v \) to \( t \) passes through \( u' \). The transitive reduction of a dominance relation produces a compact tree representation:

Definition 2: For a CFG \( C \), its pre-dominator tree \( T_{\text{pre}} = (V_C, E_{\text{pre}}, s) \) is a tree where the parent of a vertex \( u \) is termed its immediate pre-dominator and is denoted \( \text{ipre}(u) \).

Definition 3: For a CFG \( C \), its post-dominator tree \( T_{\text{post}} = (V_C, E_{\text{post}}, t) \) is a tree where the parent of a vertex \( u \) is termed its immediate post-dominator and is denoted \( \text{ipost}(u) \).

The vertices that a vertex almost dominates are those in its dominance frontier:

Definition 4: For a vertex \( u \), its post-dominance frontier, denoted \( \text{DF}_{\text{post}}(u) \), is the set \( \{v : v \text{ post-dominates some } s \in \text{succ}(v) \text{ but does not post-dominate } v\} \).

Any vertex \( h \in V_C \) that pre-dominates the source of an edge \((t, h)\) is termed a loop header; the edge itself is termed a loop-back edge and the source is called a tail. The loop-back edge identifies a subgraph of the CFG, the loop body, which is composed of the vertices that can reach \( h \) without passing through \( h \) [6]. Since \( h \) can be the destination of multiple loop-back edges, all the subgraphs induced by the loop-back edges are unioned together to create a single loop — we denote this union by \( \text{body}(h) \).

We are often interested in three other sets associated with \( h \): its tail set, \( \text{tails}(h) = \{t : (t, h) \text{ is a loop-back edge}\} \); its exit set, \( \text{exits}(h) = \{u : (u, v) \in E_C \text{ and } u \in \text{body}(h) \text{ and } v \notin \text{body}(h)\} \); and its entry set, \( \text{entries}(h) = \{u : (u, v) \in E_C \text{ and } u \notin \text{body}(h) \text{ and } v \in \text{body}(h)\} \).

A loop can then be categorised as follows:

Definition 5: A do-while loop satisfies \( \text{exits}(h) = \text{tails}(h) \); a for loop satisfies \( h \in \text{exits}(h) \) and \( |\text{exits}(h)| = 1 \); otherwise it is a break-inclusive loop.

If by removing all of the loop-back edges from a CFG \( C \) the resultant CFG \( C' \) is a DAG, we say that \( C \) is reducible. In a reducible CFG, headers pre-dominate other headers, and a nesting hierarchy therefore exists among loops:

Definition 6: For a reducible CFG \( C \), its Loop-Nesting Tree (LNT) \( T_L = (V_{L_L} = V_C \cup V_L, E_{L_L}, r, V_{L_R}, f_L) \) is a tree with the following properties:

- \( V_C \) is the set of leaves and \( V_{L_R} \subset V_C \) is the set of headers.
• \( V_L \) is the set of internal vertices, each of which represents a loop. In particular, the leaf descendants of some \( L \in V_L \) are the vertices in \( V_C \) that are contained in its body. The proper ancestors of \( L \) are the loops in which \( L \) is nested.

• The bijective function \( f_L : V_L \to V_H \) maps a loop \( L \) to its header vertex \( h \). For convenience, we use \( L_h \) to refer to the loop vertex \( L \) satisfying \( f_L(L) = h \).

• \( E_{TL} = \{(L, v) : L \in V_L, v \in V_C \} \) and

\[
\text{L is the innermost enclosing loop containing } v
\]

\[
\bigcup \{(L_h, L_{h'}) : L_h, L_{h'} \in V_L \text{ and } h \neq h' \text{ is the closest pre-dominator to } h' \text{ among } V_H\}
\]

Construction of the SB-CFG requires the acyclic subgraph of a loop to be induced such that inner loops at the next nesting level are only represented by abstract vertices. Formally:

**Definition 7:** For a loop \( L_h \), its \textbf{Induced Control Flow Graph} (ICFG) \( C_h = (V_{C_h}, E_{C_h}, s_h, t_h) \) is a DAG such that:

• \( V_{C_h} = \text{suc}(L_h). \) Any vertex \( s \in \text{suc}(L_h) \) satisfying \( s \in V_C \) is considered as an \textbf{abstract loop vertex} since it effectively represents the collapsed inner loop.

• \( E_{C_h} = \{(u, v) : u, v \in V_C \} \) and

\[
\bigcup \{(u, L) : u \in V_C, L \in V_L \text{ and } (\exists (u, v) \in E_C)(\text{parent}(v) = L)\}\]

\[
\bigcup \{(L, v) : L \in V_L, v \in V_C \} \text{ and } (\exists (u, v) \in E_C)(\text{parent}(u) = L)\}
\]

\[
\bigcup \{(L_1, L_2) : L_1, L_2 \in V_L \text{ and } (\exists (u, v) \in E_C)(\text{parent}(u) = L_1 \wedge \text{parent}(v) = L_2)\}\}
\]

• \( s_h = h \).

• When \( |\text{tails}(h)| = 1 \), this singleton vertex is the exit vertex \( t_h \); otherwise, we insert a dummy vertex \( t' \) and \( t_h \) and add edges \( \{(t, t') : t \in \text{tails}(h)\} \).

### III. INTEGER LINEAR PROGRAM FROM THE CFG

In this section we describe how an ILP is derived from a CFG \( C \) as presented by Puschner and Schedd [2]. Although Li and Malik [1] are generally credited with the original formulation, Puschner and Schedd presented a more rigorous theoretical model based on the circulation problem [3]. Moreover, they exposed the need for relative capacity constraints, which are the subject of Section V. Note that Puschner and Schedd used a variation of the CFG called the T-graph in which code appears on edges rather than in vertices (basic blocks) — we adhere to the CFG representation, though the choice is one of exposition and not substance.

The basic ILP proposed by Puschner and Schedd consists of six components:

1) An objective function.
2) Non-negativity constraints.
3) Program structure constraints.
4) Capacity constraints.
5) Relative capacity constraints. We delay discussion of these until Section V.
6) Infeasible path constraints. For space reasons, we do not discuss these in this paper, though it is important to stress that they are not mandatory to obtain a safe WCET estimate.

**The Objective Function:** In general terms, a solution to an ILP minimises or maximises an objective function composed of \( v \) decision variables, subject to a number of constraints that must be satisfied simultaneously. For the problem of WCET estimation, the ILP minimises the following objective function:

\[
Z = \sum_{v \in V_C} w(v) \cdot f(v) + \sum_{u \rightarrow v \in E_C} 0 \cdot f(u \rightarrow v) \tag{1}
\]

where \( Z \) is the returned WCET estimate, \( w(v) \) is the given WCET of a basic block \( v \), and \( f(v) \) (respectively \( f(u \rightarrow v) \)) is a non-negative execution count of \( v \) (respectively an edge \( u \rightarrow v \) ) that is set by the ILP solver. The WCET of \( s \) and \( t \) is zero. Note that edges of the CFG account for zero cost. Normally they are omitted from the objective function, but we find including them useful when comparing against the ILP of Section VI.

**Non-Negativity Constraints:** The non-negative constraints state that the execution count of basic blocks and edges must never be negative. That is, \( f(v) \geq 0 \), for all \( v \in V_C \), and \( f(u \rightarrow v) \geq 0 \), for all \( u \rightarrow v \in E_C \).

**Program Structural Constraints:** Derived directly from the CFG, these constraints represent the basic properties of program structure, intuitively stating that flow into a vertex equals flow out:

\[
\forall v \in V_C : \sum_{p \in \text{pred}(v)} f(p \rightarrow v) = f(v) = \sum_{s \in \text{succ}(v)} f(v \rightarrow s) \tag{2}
\]

**Capacity Constraints:** As Li and Malik affirmed, the maximisation of Equation (1) is \( \propto \) because each \( f(v) \) can be assigned the value \( \infty \). Capacity constraints are therefore needed which bound both the minimum and maximum execution count of each basic block. These capacity constraints are functions \( b : V_C \to \mathbb{R} \) and \( c : V_C \to \mathbb{R} \) such that:

\[
\forall v \in V_C : b(v) \leq f(v) \leq c(v) \tag{3}
\]

For a CFG, the capacity constraints of each \( v \in V_C \) are defined as follows:

\[
b(v) = \begin{cases} 
1 & \text{if } v = s, \\
0 & \text{otherwise}
\end{cases} \tag{4}
\]

\[
c(v) = \begin{cases} 
1 & \text{if } v = s, \\
b_{\text{max}}(v) & \text{otherwise}
\end{cases} \tag{5}
\]

where \( b_{\text{max}}(v) \) represents the maximum number of executions of a basic block in a single execution of the program. The upper capacity constraint on \( s \) is 1 to indicate that the path through the program is executed once.
The ILP of the previous section consists of many constraints that, in effect, propagate the upper capacity constraint of a vertex to other vertices. For instance, consider Fig. 1, which gives an example CFG and its ILP, assuming unit costs for the WCETs of basic blocks. All but one of the constraints are structural constraints. However, simple observation of the CFG structure informs us that most of these are redundant because \( s_1, v_1, v_4, v_7, t_1 \) must be assigned the same execution count. That is, once execution begins at \( s_1 \), it must flow through \( v_1, v_4, v_7 \) before finally reaching \( t_1 \). Yet this valuable information is hidden in the ILP through the structural constraints, the side effect of which is more constraints and variables.

In this section, we present a data structure that allows these sorts of structural observations to be deduced and then directly encoded in the ILP, shrinking the size of the ILP as a consequence.

Our data structure is based upon super blocks [5]:

**Definition 8:** For a CFG \( C \), its vertices \( V_C \) can be partitioned into \( n \) disjoint subsets \( s_1, s_2, \ldots, s_n \) such that, for any \( u, v \in s_i \), either \( u \) pre-dominates \( v \) and \( v \) post-dominates \( u \), or \( u \) post-dominates \( v \) and \( v \) pre-dominates \( u \). Each subset \( s_i \) is termed a **super block**.

The pertinent property of a super block, therefore, is that execution of one basic block implies execution of all other basic blocks in that set. However, this information alone is not sufficient to conclude that basic blocks in the same super block share the same capacity constraint; for example, a CFG of sequentially composed loops (i.e. without forward branches) produces a single super block, but each basic block is likely bounded by a different capacity constraint according to the loop in which it is contained. We overcome this problem by decomposing the CFG into a series of ICFGs and forming super blocks out of these acyclic regions.

After that, we insert edges between super blocks provided their execution counts are related in some way, thus producing the super block control flow graph:
Having identified super blocks in the body of a loop, the next step is to link them together according to how its loop bound propagates to other super blocks. To achieve this, we iterate through each vertex \( u \) in the ICFG (Line 8) and pick out those where control diverges (Lines 9) or converges (Lines 18). (Vertices with a single successor and predecessor are ignored because they do not affect how execution counts split and re-converge.)

Recall that, when \( u \) is a branch vertex, its execution count is effectively siphoned off to its successor edges, thereby preserving the flow conservation property in the CFG. To maintain this property in the SB-CFG, we want to represent how the super blocks containing each \( v \in \text{succ}(u) \) (Line 11) can gain execution count from the super block containing \( u \). There are two cases to consider. When \( v \neq \text{iPost}(u) \), \( v \) is control dependent [11] on \( u \), and this possible transfer of control is modelled analogously by inserting an edge between their respective super blocks (Lines 12–14). On the other hand, when \( v = \text{iPost}(u) \), we need a way of representing the fact that execution can pass directly to \( v \) without the execution count being accounted for in another basic block. To handle this situation, we insert a dummy super block (with an empty set of basic blocks), and link the super blocks accordingly (Lines 15–17). An example of a CFG with this latter property appears in Fig. 3a. Notice how the edge \( v_{10} \rightarrow v_{11} \) transfers control from \( v_{10} \) to its immediate post-dominator; the SB-CFG on the right therefore contains an empty super block whose sole predecessor is the super block containing \( v_{10} \).

Now consider when \( u \) is a merge vertex. Observe that, when \( u \) post-dominates \( \text{iPre}(u) \), the execution count that dissipates at \( \text{iPre}(u) \) must re-converge at \( u \); that is, \( \text{iPre}(u) \) and \( u \) are in the same super block and no edge insertions are needed. When this property does not hold, however, there are two possibilities.

Either \( |DF_{\text{post}}(u)| = 1 \) and this singleton vertex is \( \text{iPre}(u) \), which indicates that the nearest branch controlling access into \( u \) is \( \text{iPre}(u) \) but that \( \text{iPre}(u) \) can avoid \( u \). Fig. 3b gives an example with these properties: \( |DF_{\text{post}}(v_{16})| = 1 \) and \( v_{12} = \text{iPre}(v_{16}) \). It would be incorrect to conclude that the execution count of \( v_{16} \) is equal to \( v_{12} \) because \( v_{12} \) can reach \( v_{17} \), its immediate post-dominator, without hitting \( v_{16} \). Rather, \( v_{16} \) only gains execution count whenever one of its predecessors, \( v_{14} \) or \( v_{15} \), executes.

The other situation is when \( |DF_{\text{post}}(u)| > 1 \), which indicates that multiple branch vertices can reach \( u \) but do not necessarily pass through it. Fig. 3c gives an example with these properties: \( DF_{\text{post}}(v_{23}) = \{v_{18}, v_{19}\} \). Here the execution count of \( v_{23} \) increases only when one of the vertices in its post-dominance frontier diverge to \( v_{21} \) or \( v_{20} \), the predecessors of \( v_{23} \).

In both cases, the super block with such a merge vertex only accumulates execution count whenever one of its predecessors in the CFG executes (Lines 18–23).

The final stage of the algorithm is to represent how execution of the super blocks obtained in a nested loop \( L_{h'} \) depend on the path taken through its parent loop \( L_h \). First observe that, when we create the portion of the SB-CFG inside \( L_{h'} \), it always generates a super block \( v \) without predecessors such that \( h' \in \sigma(z) \). (The vertex \( z \) cannot have predecessors because \( h' \) does not have predecessors in the ICFG \( C_{h'} \)). Furthermore, \( h \notin \sigma(z) \) since the outer nested loops of \( h' \) are irrelevant to the construction of \( C_{h'} \). The vertex \( z \) is therefore the unlinked super block which we wish to connect. Further note that \( L_{h'} \) is a vertex in \( C_h \), in effect representing its reduction, and therefore \( L_{h'} \) is in another super block \( y \neq z \). Only when \( y \) is executed does \( z \) and its descendant super blocks accumulate execution count: thus we add an edge \( y \rightarrow z \) to represent this dependence (Lines 24–26). An example of a SB-CFG for a cyclic CFG appears in Section V.

**Comparison with Agrawal**: Agrawal has also proposed a data structure based on super blocks. The most obvious distinction is that he recognises super blocks in the CFG as a whole, rather than decomposing down to individual ICFGs. Moreover, the way in which he inserts edges differs considerably: an edge \( y \rightarrow z \) is added provided there is a basic block in \( \sigma(y) \) that pre- or post-dominates a basic block in \( \sigma(z) \). However, this does not model the same information as the SB-CFG because these dominator edges might not have corresponding edges in the CFG.

**Handling break Constructs**: We assume that CFG loops are either do-while or for constructs and not break-inclusive loops (see Definition 5). This is because, otherwise, we would have to include the exit path out of inner loops inside the ICFG of the enclosing loop. This assumption prevents that because, with only a single exit out of the loop in its head or through a tail, it suffices to represent inner loops as single abstract vertices: in a do-while loop the upper capacity constraint of all vertices is the same; in a for loop, the header can iterate a number of times more that its body. Although it is trivial to add support for break-inclusive loops in the SB-CFG, we
cannot yet claim this leads to a reduced constraint system.

**V. RELATIVE CAPACITY CONSTRAINTS**

The ILP discussed in Section III returns an upper bound on the actual WCET provided the execution times and upper capacity constraints on the decision variables are safe. As proven by Puschner and Schedl, however, overestimation is possible because the ILP does not always precisely characterise the set of execution paths through the CFG.

The crux of the problem is that the ILP solver can assign an upper capacity constraint to a loop header’s execution count while still assigning zero execution counts to edges entering the loop. More formally, let $C' = (V_C, E_C, s, t)$, where $E_C' = \{(u, v) \in E_C : f(u \to v) > 0\}$, be the circulation subgraph induced by the ILP solver through a CFG $C$. Then we say $C'$ contains a disconnected circulation if it is not strongly connected.

We illustrate the problem through Fig. 4a, which is identical to the T-graph used by Puschner and Schedl in their discussion of this problem. The CFG has three loop-back edges: $v_{29} \to v_{29}, v_{28} \to v_{27},$ and $t_2 \to s_2$; these induce respective loops $L_{v_{29}}, L_{v_{27}},$ and $L_{s_2}. Assume that the upper capacity constraints of $v_{29}$ and $v_{27}$ are 45.

Fig. 4b gives the ILP, a solution of which returns execution counts of 45 for $v_{29}, t_2, v_2,$ and 0 for both $e_{14}$ and $e_{17}$. In effect, the loop $L_{v_{29}}$ is disconnected from the path through its innermost enclosing loop $L_{s_2}.

The solution as proposed by Puschner and Schedl is to include relative capacity constraints. These model the execution count of a loop with respect to its parent loop, rather than, or in addition to, its upper capacity constraint. Relative capacity constraints require, for a loop $L_h$, its set of implicating edges $E_{imp} \subset E_C$, which are those whose sources are entries$(h)$.

Formally, therefore, a relative capacity constraint on $L_h$ is:

$$f(h) \circ \sum_{u \leftarrow v \in E_{imp}} k \cdot f(u \to v)$$

with $\circ \in \{<, \leq, =\}$ and at least one $k$ is greater than zero.

The set of implicating edges of $L_{v_{27}}$ is $\{e_{14}\}$. Transforming (28) into the following relative capacity constraint:

$$v_{27} \leq 45 \cdot e_{14}$$

returns an accurate WCET estimate such that $e_{14}, e_{17}$ have non-zero execution counts.

Puschner and Schedl claim that relative capacity constraints are needed for every loop, except for the loop induced by the loop-back edge $t \to s$. Yet the reader will notice that there was no requirement to add a relative capacity constraint on $L_{v_{29}}$ in spite of the fact it is at the same nesting level as $L_{v_{27}}.

By observation of the CFG, this is because when execution begins at $s_2$ it must always pass through $v_{29}$; on the contrary, the execution of $v_{27}$ is control dependent on $v_{29}$ and it can therefore be skipped. The SB-CFG allows us to deduce this information:

**Theorem 1:** For distinct headers $h, h'$ such that $L_h$ is the innermost enclosing loop of $L_{h'},$ a relative capacity constraint is not needed on $h'$ provided its abstract loop vertex $L_{h'}$ is in the same super block as $h$.

**Proof:** From the assumption of reducibility, it follows that $h$ predominate $h'. Because h, L_{h'} \in \sigma(y),$ it follows from Definition 8 that $L_{h'}$ must post-dominate $h.$ By the definition of post-dominance, each time $h$ executes so too does the loop $L_{h'}.$ As such, the loop-entry or loop-exit edges of $L_{h'}$ can only be assigned values of zero by the constraint system solver if $h$ is assigned zero; therefore a relative capacity constraint is not needed.

For illustrative purposes, let us apply this theorem to the problematic CFG. Fig. 4c depicts the corresponding SB-CFG in which edges added between the super blocks of nested loops are marked. In this example, there are two such edges, linking super blocks formed in the outer loop $L_{s_{2}}$ to super blocks formed in the inner loops $L_{v_{29}}$ and $L_{v_{28}}.$ Note that $L_{v_{29}}$ shares a super block with $s_2,$ the header of its innermost enclosing loop — from Theorem 1, $L_{v_{29}}$ does not need a relative capacity constraint. On the other hand, $L_{v_{28}}$ does not share a super block and, consequently, a relative capacity constraint is mandatory to guarantee a tight WCET estimate.

In this example, the coefficients of the capacity constraint and the relative capacity constraint are equal because there is only one level of loop nesting. However, this does not generally hold: the capacity constraint models how many times a header can execute globally, whereas the relative capacity constraint represents a scoped loop bound. Thus the benefit of inferring which loops only require capacity constraints is twofold. First, it eliminates potentially unnecessary constraints. Second, loop bound extraction techniques [12–14] can concentrate on the global execution count.

**VI. INTEGER LINEAR PROGRAM FROM THE SB-CFG**

The primary motivation for the SB-CFG is to reduce the size of the constraint system. With this goal in mind, this section shows how to derive an ILP from a SB-CFG.

**A. Variables and Objective Function**

Since the SB-CFG and the CFG both model the same unit of computation, namely the basic block, both ILPs always have a variable for each $v \in V_C.$ The principal difference, therefore, is how these variables are related. The CFG model introduces variables for each edge $(u, v) \in E_C$ to relate how basic block variables are constrained. In a similar vein, the SB-CFG also needs another set of variables, $X,$ to relate execution counts across super block boundaries. The set $X$ is in fact the union of three other sets.

First, a variable is required for every abstract loop vertex included in the super blocks. However, since abstract loop vertices are not covered by the variables for $V_C,$ we introduce one extra for each $L \in V_L.$ This set is denoted $X_V.$

Second, every vertex in the SB-CFG needs a variable representation in the ILP. Any vertex $y \in V_S$ satisfying $\sigma(y) = \emptyset,$ however, is neither covered by the variables for $V_C$ nor $X_V$, and thus we introduce a variable for each such $y.$ This set is denoted $X_y.$
Lastly, we need a variable for every edge \((y, z) \in E_2\) such that \(|\text{succ}(y)| > 1\) and \(|\text{pred}(z)| > 1\). Reasons for these variables become clear in the discussion below. This set is denoted \(X_{E_2}\).

**Objective Function:** The objective function is similar to that of (1):

\[
Z = \sum_{v \in V_G} w(v) \cdot f(v) + \sum_{x \in X_{V_6}} w_L(x) \cdot f(x) + \sum_{x \in X_{E_6} \cup X_{E_2}} 0 \cdot f(x) \quad (32)
\]

where the function \(w_L\) returns the WCET of an abstract loop vertex \(L_h\). In the case of a for loop, this value is the WCET value of the header \(h\) because, as noted above, the header of a for loop can execute a number of times extra than its body. On the other hand, in a do-while loop, this value is zero because its execution time is accounted for entirely in its super blocks.

The other variables \(X_6 \cup X_{E_2}\) contribute zero cost to the objective function because they have no execution cost, analogously to CFG edges in the ILP of Section III.

**B. Constraints**

Before we describe how constraints are generated, we need some additional notation. Specifically, we need to differentiate the successors of a super block between those added during analysis of an ICFG and those added during construction of the SB-CFG. Thus, for a vertex \(y \in V_{S}\), we define \(\text{succ}'(y) = \{s \in \text{succ}(y) : (\exists h \in \sigma(s))(h \in V_{ilp})\}\).

Constraints in the SB-CFG ILP are specific to the properties of each super block. In particular, we identify the following subsets of \(V_S\):

- **Super blocks composed of multiple vertices:**
  \[V_{S,1} = \{y : |\sigma(y)| > 1\}\]

- **Branch super blocks whose successors do not contain merge super blocks:**
  \[V_{S,2} = \{y : |\text{succ}'(y)| > 1\} \text{ and } (\exists s \in \text{succ}'(y))(|\text{pred}(s)| > 1)\]

- **Merge super blocks whose predecessors do not contain branch super blocks:**
  \[V_{S,3} = \{y : |\text{pred}(y)| > 1\} \text{ and } (\exists p \in \text{pred}(y))(|\text{succ}'(s)| > 1)\]

- **Other branch and merge super blocks:**
  \[V_{S,4} = \{y : |\text{succ}'(y)| > 1\} \text{ and } (\exists s \in \text{succ}'(y))(|\text{pred}(s)| > 1)\] \[\cup \{y : |\text{pred}(y)| > 1\} \text{ and } (\exists p \in \text{pred}(y))(|\text{succ}'(s)| > 1)\]

- **Super blocks which contain a header vertex:**
  \[V_{S,5} = \{y : (\exists h \in \sigma(y))(h \in V_{ilp} \text{ and } h \neq s)\}\]

The SB-CFG ILP uses a representative variable of a super block to link constraints together:

**Definition 10:** For a vertex \(y \in V_{S}\), its representative variable, denoted \(v_{y}^{rep}\), is a (randomly selected) variable of a vertex \(v \in \sigma(y)\) if \(\sigma(y) \neq \emptyset\) or a distinct \(x_y \in X_6\) otherwise.

**Intra-Super-Block Constraints:** These constraints express that the constituent vertices of a super block share the same execution count. Thus they only apply to the set \(V_{S,1}\):

\[
\forall y \in V_{S,1}, \forall v \in \sigma(y) \setminus \{v_{y}^{rep}\} : f(v) = f(v_{y}^{rep}) \quad (33)
\]

A further constraint is required on the super block containing \(s\), denoted \(y_s\), which is analogous to (5):

\[
f(v_{y_s}^{rep}) = 1 \quad (34)
\]

**Inter-Super-Block Constraints:** These constraints are case split depending on the properties of super blocks.

Case 1: the set \(V_{S,2}\). Every \(y \in V_{S,2}\) must contain a branch basic block. However, different super block successors of \(y\) might be controlled by different branch vertices in \(y\). Hence we define an equivalence relation \(\sim\) on \(\text{succ}'(y)\) where \(z \sim z'\) if, and only if, the edges \((y, z)\) and \((y, z')\) were added due to the same branch basic block \(u \in \sigma(y)\). (Recalling Fig. 2, this is either because \(u\) controls entry into some \(v \in \sigma(z)\) or \(v' \in \sigma(z')\), or because there is an edge \((u, \text{ipost}(u)) \in E_C\) and either

![Diagram](image-url)
For each equivalence class [z], only one of its super block destinations can be executed for every execution of y:

$$\forall y \in V_{S,2}, \forall [z] \in succ'(y)/\sim : f(v_{y}^{re^p}) = \sum_{z \in [z]} f(v_{x}^{re^p})$$  \hspace{1cm} (35)

Case 2: the set $V_{S,3}$. Every $y \in V_{S,3}$ must contain a merge basic block that does not post-dominate its immediate pre-dominator. Thus $y$ accumulates execution count whenever one of its predecessors executes:

$$\forall y \in V_{S,3} : f(v_{y}^{re^p}) = \sum_{p \in pred(y)} f(v_{p}^{re^p})$$  \hspace{1cm} (36)

Case 3: the set $V_{S,4}$. We separate out these branch and merge super blocks as generating their constraints demands careful analysis. That is, blindly applying (35) and (36) to these super blocks would eliminate particular feasible paths from the calculation. An example will clarify.

$$\forall y \in V_{S,4} : f(v_{y}^{re^p}) = \sum_{p \in pred(y)} f(v_{p}^{re^p})$$  \hspace{1cm} (37)

Substituting the right-hand side of (38) into (37) yields:

$$v_{33} = v_{35} + v_{36} + v_{34}$$  \hspace{1cm} (38)

The $v_{33}$ terms cancel each other out:

$$v_{35} + v_{34} = 0$$  \hspace{1cm} (39)

In effect, the ILP solver can never assign an execution count larger than zero for $v_{34}$ or $v_{35}$, irrespective of their WCETs.

Fig. 5 shows a CFG on the left and its respective SB-CFG on the right. We see that $V_{S,4} = \{S_2, S_3\}$. Applying (35) and (36) to these super blocks gives:

$$v_{33} = v_{35} + v_{36}$$  \hspace{1cm} (40)

Substituting the right-hand side of (38) into (37) yields:

$$v_{33} = v_{35} + v_{36} + v_{34}$$  \hspace{1cm} (38)

The $v_{33}$ terms cancel each other out:

$$v_{35} + v_{34} = 0$$  \hspace{1cm} (39)

In effect, the ILP solver can never assign an execution count larger than zero for $v_{34}$ or $v_{35}$, irrespective of their WCETs.

To prevent this situation, we instead attach a variable $x_{y,z} \in X_{E_S}$ to each $(y, z) \in E_S$ satisfying $|succ(y)| > 1$ and $|pred(z)| > 1$. Because the set $V_{S,4}$ includes both branch and merge super blocks, we need to handle each differently.

For a branch super block $y$, we further partition each of its equivalence classes $[z] \in succ'(y)/\sim$ into two sets: $[z]_1$ contains all super blocks with a single predecessor, while $[z]_2$ contains the remaining super blocks. Therefore:

$$\forall y \in V_{S,4}, \forall [z] \in succ'(y)/\sim : f(v_{y}^{re^p}) = \sum_{z \in [z]} f(v_{z}^{re^p}) + \sum_{z \in [z]_2} f(x_{y,z})$$  \hspace{1cm} (41)

Similarly, for a merge super block $y'$, we split its predecessors into two disjoint subsets: $pred_1(y')$ contains all super blocks with a single successor, while $pred_{>1}(y')$ contains the remaining super blocks. Therefore:

$$\forall y \in V_{S,4}, \forall [z] \in succ'(y)/\sim : f(v_{y}^{re^p}) = \sum_{z \in [z]} f(v_{z}^{re^p}) + \sum_{z \in [z]_2} f(x_{y,z})$$  \hspace{1cm} (42)

Observe from Fig. 2 that all proper ancestors of $h$ in $T_L$, except $L_h$, must be in a super block on some path from $r_S$, the root of the SB-CFG, to $y$. Let $A_y$ denote this set of super blocks and define a function $f_{A_y} : A_y \rightarrow V_L$ which returns the ancestor loop vertex of a given super block. Then:

$$\forall y \in V_{S,5} : f(v_{y}^{re^p}) = k \cdot f(v_{y}^{re^p})$$  \hspace{1cm} (43)

where $a \in A_y$, $a \in \{<, \leq, =\}$, and, for the unique header $h \in \sigma(y), k = f_{bound}(h, f_{L}(f_{A_y}(a))).$ (Recall from Definition 6 that $f_L$ returns a header vertex in the CFG from an abstract loop vertex.)

VII. EVALUATION

Our hypothesis is that the constraint system derived from the SB-CFG is smaller than that obtained from the CFG and, crucially, leads to faster solution times.

A. Experimental Set Up

The goal of our experiments was to observe how the properties of the constraint system (i.e. number of variables/ constraints and solving time) change with CFGs of differing sizes (i.e. number of vertices) and compositions (i.e. edges).

With a benchmark suite, it is impossible to control these parameters as the structural properties of a CFG are already determined by the source code and how the compiler generates code. Furthermore, the objective of our experiments was not to compute WCET estimates for actual programs and justify their accuracy, since this demands a complete WCET toolchain [15]. For these reasons, we developed a tool which automatically generates synthetic CFGs.

CFG Generation: The tool provides control over the following parameters with respect to the CFG: its number of vertices; the maximum number of successors of a vertex (default is two); its number of loops (default is zero); the maximum loop-nesting depth; and whether a loop has multiple tails (mimicking continue-like statements).

For an acyclic CFG, the generation proceeds by randomly choosing how many subgraphs it wishes to generate of a particular structure, under the constraint of the number of chosen vertices. The structures of choice are pre-defined, typical code fragments found in CFGs, of which there were four choices: 1) A structured if-then-else construct with a branch vertex $b$ and a merge vertex $m$ such that $b = ipre(m)$,
\( m = \text{ipost}(b) \). In addition, \( b \) branches to two distinct successors which eventually re-converge at \( m \); 2) A structured if-then-else construct, which is similar to the if-then-else construct except one of the distinct successors of \( b \) is \( m \); 3) An unstructured if-then-else construct which is analogous to that depicted in Fig. 3c; 4) A structured switch-like construct, provided the maximum number of successors exceeds two. Again this is similar to the if-then-else, except the branch vertex \( b \) has \( n > 2 \) distinct successors.

During the construction of one of these structures, there will be \( n \geq 0 \) previously generated structures that remain disconnected. To synthesise nested constructs, therefore, we randomly decide whether to link a disconnected component when building a path from a branch to a specific merge vertex. If there remains disconnected components on completion of this subgraph generation phase, we merely chain these components together.

This is sufficient to generate a CFG compliant with Definition 1. To inject a further amount of randomness into the CFG structure, however, two additional tasks were undertaken. First, we removed certain merge vertices that post-dominate their branches. The rationale is that, typically, there will be more branch points in a CFG than merge points, since nested if-then-else constructs need not re-converge at different points. Second, we added edges \( u \rightarrow v \) such that \( v \) appears after \( u \) in a topological sort and \( |\text{succ}(u)| = 1 \). The rationale is that these edges add unstructuredness to the CFG, a property more prevalent in CFGs obtained from object code.

When the user chooses to generate a cyclic CFG, we created an artificial LNT that respects the chosen number of loops and loop-nesting depth. Then we effectively reverses the CFG decomposition used in Section IV by creating an acyclic graph for each loop in this LNT, using the method outlined above. The start and exit vertices of this acyclic graph became the loop header and tail, respectively.

Every loop is nested inside its parent loop by choosing either the header vertex \( h \) (for loop) or the tail vertex \( t \) (do-while loop) as the source of the exit. When continue-like statements have been requested, we additionally chose a vertex \( v \in \text{body}(h) \) such that \( |\text{succ}(v)| = 1 \) and \( v \notin \text{tails}(h) \); we then add the edge \( v \rightarrow h \).

**ILP Generation:** We have implemented the ILPs of Sections III and VI in another prototype tool which consumes a synthetic CFG. Since the CFG is artificially created, there is no possibility of gathering the data vital to the WCET calculation, such as the WCET of basic blocks and loop bounds, through simulation or likewise. To overcome this deficiency, we assigned unit WCETs to basic blocks and loop structures. To synthesise nested constructs, therefore, we randomly decide whether to link a disconnected component when building a path from a branch to a specific merge vertex. If there remains disconnected components on completion of this subgraph generation phase, we merely chain these components together.

**The Toolchain:** These tools were run in tandem as follows:\(^1\)

First, we had a script that iterated through the range \([100, 2000]\), where each index \( i \) is the number of vertices to include in the automatically generated CFG. The range is motivated in part by previous experiments: a low number of vertices does not stress the ILP solver, whereas a too high number leads contrarily to exorbitant solving times. Perhaps more so, we believe this range encapsulates the size of CFGs encountered in practice. For each value of \( i \), we produced 3 different variations of the synthetic CFG, i.e. the edge sets change. Motivation for this choice stems from the fact that an ILP could, in principle, be pre-disposed to a particular CFG structure; by varying the CFG, the likelihood diminishes.

Given a CFG, we generated the ILPs as described above and then solved them via the ip_solve library [16]. The solver was in fact called 3 times since we had previously observed anomalous times due to interference from background activities of the operating system. For each ILP, we recorded the number of variables, the number of constraints, and the time taken to solve. Furthermore, for validation purposes, the tools checked that the WCET estimates returned by the different ILPs were equal.

**B. Results**

All the measurements given in this section were obtained on a PC running Ubuntu 10.04 in a virtual environment with a 3.6 Ghz Intel Core 2 Duo and 8 GiB of RAM. Our results are presented in Fig. 6–8.

Fig. 6 plots the number of constraints in the CFG ILP and the SB-CFG ILP against the number of vertices in the CFG. Specifically, since we generate 3 variations of a CFG, and each ILP has \( c_i \) constraints, the value of the y-coordinate is \( \left\lceil \frac{c_0 + c_1 + c_3}{3} \right\rceil \).

The immediate conclusion is that the SB-CFG ILP generates many fewer constraints. For example, for a CFG containing 2000 vertices, we see that there are \( \approx 2000 \) constraints in the SB-CFG ILP, contrasted with \( \approx 4000 \) in the CFG ILP. This leads us to conjecture that, in general, the SB-CFG and CFG ILPs produce \( |V_C| \) and \( 2 \cdot |V_C| \) constraints, respectively, although we do not yet have a formal proof. Overall, the total number of constraints in the CFG ILP is 3,888,313, whereas...
in the SB-CFG ILP it is $1,740,013$ — this is a $55\%$ reduction.

Fig. 7 plots the number of variables utilised in both ILPs against the number of vertices in the synthetic CFG. As expected, fewer constraints go hand in hand with fewer variables since there are not as many entities in the ILP to bound. Overall, the total number of variables in the CFG ILP is $4,331,161$, whereas in the SB-CFG ILP it is $2,184,762$ — this is a $50\%$ reduction.

Fig. 8 plots the time consumed in solving each ILP, measured in microseconds by the lp_solve library. Since there are $3$ variations of each CFG and we solve each ILP $3$ times, the $y$-coordinate in this case is $\sum_{i=1}^{3} \sum_{j=1}^{3} \text{time}_{i,j}$, where $\text{time}_{i,j}$ denotes the ILP solving time for variation $i$ and run $j$. The curves follow a similar pattern, but the SB-CFG ILP always leads to a faster solution. Overall, the total time consumed in the solver for the CFG ILP is $851,468,052,802$ microseconds, whereas in the SB-CFG ILP it is $261,849,352,312$ microseconds — this is a $69\%$ reduction.

Our experiments clearly show, therefore, that the SB-CFG produces a smaller constraint system and, consequently, a faster solution to the WCET problem.

VIII. CONCLUSIONS

This paper has presented a new intermediate data structure, the Super Block Control Flow Graph (SB-CFG). Its principal usage is to reduce the number of variables and constraints present in the constraint system that models WCET estimation. By reducing the size of the model, this work facilitates the addition of other constraints that are more pertinent to reducing the pessimism in the WCET estimate. Our practical experiments confirmed that the size of the constraint system is indeed smaller, significantly so in some cases, across a wide range of control flow graphs with diverse structural properties.

We believe that the SB-CFG potentially has other uses in terms of WCET calculations. For instance, it is possible to develop a tree-based calculation engine which would provide low computational complexity: this is the subject of our future work.

REFERENCES