Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2008-4

Partial Program Admission by Path Enumeration

Authors: Michael Wilson, Ron Cytron, Jon Turner

March 10, 2008

Corresponding Author: mlw2@arl.wustl.edu

Abstract: Real-time systems on non-preemptive platforms require a means of bounding the execution time of programs for admission purposes. Worst-Case Execution Time (WCET) is most commonly used to bound program execution time. While bounding a program's WCET statically is possible, computing its true WCET is difficult without significant semantic knowledge. We present an algorithm for partial program admission, suited for non-preemptive platforms, using dynamic programming to perform explicit enumeration of program paths. Paths - possible or not - are bounded by the available execution time and admitted on a path-by-path basis without requiring semantic knowledge of the program beyond its Control Flow Graph (CFG).

Partial Program Admission by Path Enumeration

Michael Wilson
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130
Email: mlw2@arl.wustl.edu

Ron Cytron
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130
Email: cytron@cse.wustl.edu

Jonathan Turner
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130
Email: jon.turner@arl.wustl.edu

***Abstract*—Real-time systems on non-preemptive platforms require a means of bounding the execution time of programs for admission purposes. Worst-Case Execution Time (WCET) is most commonly used to bound program execution time. While bounding a program's WCET statically is possible, computing its true WCET is difficult without significant semantic knowledge. We present an algorithm for *partial program admission,* suited for non-preemptive platforms, using dynamic programming to perform explicit enumeration of program paths. Paths – possible or not – are bounded by the available execution time and admitted on a path-by-path basis without requiring semantic knowledge of the program beyond its Control Flow Graph (CFG).**

I. Introduction

Admission control in real-time systems running on non-preemptive platforms requires the ability to bound the execution time of applications. In a trusted environment, a single administrator can make an out-of-band determination of execution boundedness. Untrusted, shared environments are more difficult. As an example of such an environment, consider network virtualization, which has been advanced as a way to foster innovation in the Internet [1].

In network virtualization, core router platforms host 3rd-party application code, running at Internet core speeds, allowing the creation of high-speed overlay services [2]. These platforms, of which the IXP 28XX is a representative example, usually have no preemption mechanism suitable for use at high speeds. Internet core speeds necessitate extremely tight cycle budgets for packet processing. To share this type of system among untrusted parties requires stringent admission control.

In other domains, instrumentation with runtime checks to enforce proper behavior is a practical solutions. Unfortunately, Internet core speeds render runtime checks impractical. At 5Gbps, an IXP 2800-based system with 1.4 GHz microengines and 8 hardware thread contexts has a compute budget of 170 cycles. With such tight budgets, even a few runtime checks can quickly push otherwise admissible program paths over budget. A practical solution must therefore impose as little runtime overhead as possible.

Worst-Case Execution Time (WCET) analysis is the currently accepted approach. A WCET bound can be established statically, assuming that all program paths are viable. However, some well behaved programs might be rejected. For example, a program may have mutually exclusive code paths that, taken together, exceed the cycle budget. Demonstrating that these paths are mutually exclusive takes semantic knowledge, either provided by the developer or deduced by analysis at admission time. In most domains, this information is provided by the developer as branch constraints. For our virtualization application, we cannot trust the developer; any semantic knowledge must come from the analysis.

We propose *partial program admission* as a practical solution to this problem. By explicitly examining all paths, we can perform static analysis to re-write 3rd-party applications to achieve the following goals:

1) all "safe" paths (paths that complete under budget) are admitted,
2) no "unsafe" paths (paths that complete over budget, or that do not complete) are admitted,
3) no runtime penalty is imposed on any safe path, and
4) no semantic knowledge is required.

To re-write the program, we actually duplicate some code paths. While this causes some code expansion, or "bloat", in practical cases the bloat proves to be within acceptable limits.

Partial program admission seems at first glance to be a useless process. It is uncommon for a developer to wish to run only some fragment of a program. However, our construction for partial program admission is not intended for running only portions of a program, but for generating a new program where the proof of execution time correctness is trivial.

WCET analysis depends upon developer knowledge of branch constraints to eliminate paths that, while present in the program, could never be taken. If the developer has met the desired budget goals, all paths that can actually be taken will be under budget. Only "impossible" paths are excluded. In this way, we allow some code duplication to substitute for detailed understanding of the program.

We also note that, during development, the program may not be under budget. The same partial admission can also serve to inform the developer of program paths that have unexpectedly run over budget. We view this algorithm as a development tool as well as an admission tool.

We present a theoretical construction that mirrors our algorithm in Section II. In section III, we present our actual algorithm, followed by proofs of correctness in Section IV. We follow up with some preliminary performance data in

Section V, related work in Section VI, and our plans for future work in Section VII.

II. Algorithm Foundations

In this section we define the theoretical constructions on which our algorithm is based. First, we describe the computational model in which our solution works. Next, we describe a series of graph transformations culminating in a construction which meets our goals at the cost of significant code duplication. Finally, we describe a means of reducing the code duplication.

These constructions form the basis of an algorithm which is functionally identical to, but intractably slower than, our algorithm.

A. Computational Model

Our algorithm should be considered in the context of a simplified processor. Our idealized processor has instructions taking exactly one cycle to complete. All memory accesses complete in one cycle. There is no pipeline. There is no preemption.

Our computational model is event-driven, where code is executed only in response to these events. For the network virtualization application, the event is packet arrival.

Each block of code must complete within some number of cycles, known at admission time. Cycles may not be "saved" from one call to the next. The guarantee we must enforce is that, from the time the code is called to the time the code returns control, it consumes no more than the cycle count, called the *budget*.

Finally, we require the developer to add a "time-exceeded" exception handler to her code. The exception handler is required to adhere to strict coding guidelines which make static analysis simple and easy.

The requirements of our model are sufficient, but not strictly necessary. Our algorithm continues to work so long as at every node we examine, we carry all of the information necessary to determine the total execution time of every path beginning at that node. For example, suppose we have a memory cache. The execution time of subsequent instructions will depend upon the contents of the cache, which can be derived from prior instructions, memory layout, and the behavior of the cache. Our model is chosen to simplify this information as much as possible.

B. Path Enumeration

Our input to the algorithm consists of an assembly level representation of the program. From this, we can develop a Control Flow Graph (CFG) of the program, in which outgoing edges are labeled by the execution time required for the corresponding program segments. Our objective is to derive a new CFG that executes the same sequence of instructions for program executions that complete within a specified time bound B, while terminating in an exception handler for program executions that exceed the budget B.

The conceptual starting point for this construction is the creation of a *Control Flow Tree* (CFT) from the CFG. The CFT duplicates nodes in the CFG as necessary, in order to convert the graph into a tree.

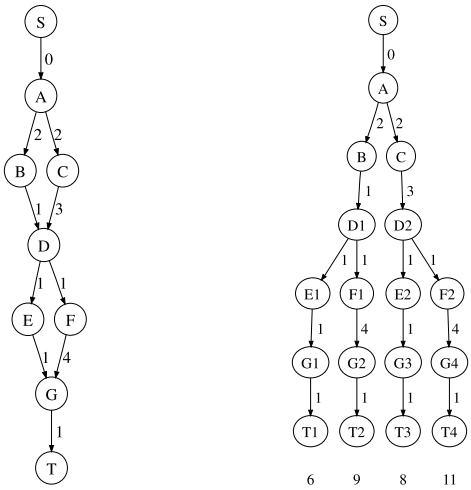

Fig. 1. CFG and the corresponding CFT. Weights along the edges represent cycle counts to traverse that edge. Total path cycle counts are presented below each terminal node in the execution tree.

See Figure 1 for an example. Nodes S and T are dummy nodes used to delineate entry and exit points, and contain no actual code. Similarly, in the CFT, $T1 - T4$ are copies of the dummy node T and contain no code.

Code generated from the CFT is functionally identical to the original CFG. If the length of the path from the root node to a node u in the tree exceeds B, then we can replace the subtree rooted at u with an exception node, representing a jump to the exception handling routine. As an additional step, if after applying this step, the CFT contains a subtree whose leaves are all exception nodes, we can replace the entire subtree with an exception node.

This pruning procedure is illustrated on Figure 1. Let us consider a budget of 10 cycles. While it would be valid to execute the path $A \rightarrow C \rightarrow D2 \rightarrow F2 \rightarrow G4$ before aborting to the exception handler, it is clear that any execution path reaching $F2$ will go over budget. Our earliest chance to raise the exception is by intercepting the branch instruction at $D2$, with the result shown in Figure 2.

We refer to the tree constructed in this way as the *B-bounded execution tree* of the original control flow graph. We note that such a tree can be defined relative to any node u in the CFG and we let $bxt_B(u)$ (or generally, BXT) denote this execution tree.

While one could generate a version of the original program directly from the BXT, this typically results in an excessive amount of code duplication. We can dramatically reduce the amount of code duplication by merging equivalent subtrees of the BXT in a systematic way.

C. Code Duplication Reduction

The BXT typically contains many subtrees that are identical to one another and can be merged. To make this precise, we define two nodes u_1 and u_2 in the BXT to be *equivalent* if they were derived from the same node u in the original CFG

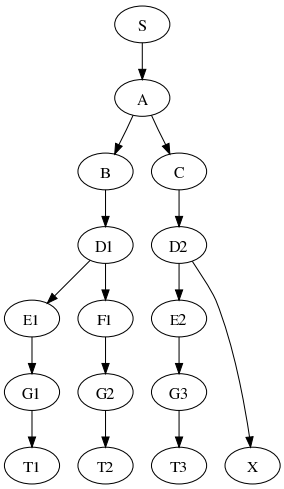

Fig. 2. Abort to exception handler

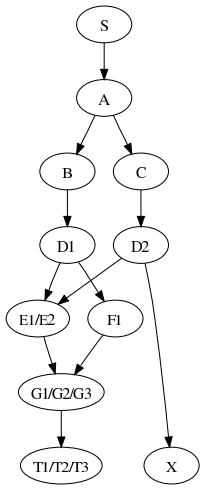

Fig. 3. Merging of equivalent execution subtrees

(that is, they represent copies of the same original program segment). Two subtrees of the BXT are equivalent if they are structurally identical and all of the corresponding node pairs are equivalent. We can merge any pair of equivalent subtrees without changing the set of executions, yielding a *bounded execution graph* (BXG) equivalent to the BXT. Conceptually, the merging is performed in a top down fashion. That is, if u_1 and u_2 are roots of equivalent subtrees, we merge them so long as there are no ancestors v_1 of u_1 and v_2 of u_2 that are also roots of equivalent subtree. The merging process continues, as long as there are equivalent subtrees that can be merged.

Returning to our example, nodes $D1$ and $D2$ cannot be merged because their child execution trees are different. $D1$ has children $E1$ and $F2$; $D2$ has children $E2$ and X. However, the subtrees rooted at $E1$ and $E2$ are identical. There is no need to retain both trees. Instead, we can merge them into a single subtree. Even further, the tree rooted at $G2$ is identical to the subtrees rooted at $G1$ and $G3$. We can also merge the $G2$ node with the $G1/G3$ node from the $E1/E2$ execution tree. See Figure 3.

In contrast to the massive code duplication in the BXT, in the BXG only one node (D) needed to be duplicated.

D. Intervals

While one can derive the BXG by explicitly constructing the BXT and then merging nodes, there is a more efficient dynamic programming procedure that can be used to construct the BXG directly. This procedure is based on the observation that the structure of a BXT subtree with root node u_1 is a function of just two things – the node u in the original CFG from which u_1 was derived and the amount of available execution time that remains after execution has reached u_1. If the cost of the path from the root to u_1 is p, then the remaining execution time is $B - p$ where B is the overall bound. We note that the BXT subtree with root u_1 is $bxt_{B-p}(u)$. So two nodes u_1 and u_2 derived from the same CFG node u will have identical subtrees if the costs of their paths from the root are identical.

We can extend this notion to path costs that are "close." Given nodes u_1 and u_2 derived from u, with path costs from the root of p and q respectively, they will have identical subtrees if $bxt_{B-p}(u) = bxt_{B-q}(u)$. This will be true for values of $B-p$ and $B-q$ that are "close enough" in a certain sense.

For each node u in the original CFG, the dynamic programming procedure produces a partition on the integers corresponding to a partition on subtrees. Two values i and j fall in the same equivalence class of the partition if and only if $bxt_i(u) = bxt_j(u)$. Using these partitions, we can construct the BXG directly from the CFG, without having to explicitly construct the BXT.

As we prove in section IV, this partition of the integers falls into contiguous ranges from a minimum value to a maximum value, and including all values between. For our algorithm, we refer to these partitions of the integers as *intervals*, and use these as the basis for a memoization scheme.

III. The Algorithm

Before we formally present the algorithm there are several preliminary details to define.

First, we assume that the code has already been read into a CFG with S and T nodes. $w(u)$ represents the cycle cost to traverse node u, represented as an outgoing edge weight in our CFGs.

We also assume that we have an INTERVAL data type. We represent each INTERVAL as a pair $[a, b]$ where $a < b$. Each INTERVAL is treated as the set $\{x | a \leq x \leq b\}$, with the usual definitions for intersection, subset, overlapping INTERVALS, disjoint INTERVALS, and element predicate ($\in$). We define scalar addition on an INTERVAL as $[a, b] + x = [a + x, b + x]$. Finally, we define the **null** INTERVAL as the empty set.

Given the INTERVAL type, we define an INTERVAL search object with two functions.

- INTERVAL **function** insert(**vertex** v, INTERVAL i)
  Adds a tuple $< v, i >$ to the search object; returns the INTERVAL.

```
INTERVAL function bxg(integer R, vertex u)
    INTERVAL i
    i := find(u,R)
    return if i ≠ null → i
        i = null and R < δ(u,T) → insert(u,[−∞,δ(u,T)-1])
        i = null and R ≥ δ(u,T) and u = T → insert(u,[0,∞])
        i = null and R ≥ δ(u,T) and u ≠ T
            → insert(u, ⋂_{v∈child(u)} (w(u,v)+bxg(v,R-w(u,v))))
    fi
end
```

Fig. 4. Pseudo-code of the algorithm (Tarjan notation)

- INTERVAL **function** find(**vertex** v, **integer** x)
  Returns the INTERVAL associated with vertex v in which x is found, or **null** if no such INTERVAL exists.

Finally, we presume that we have pre-computed the shortest paths from each node to T by Dijkstra's algorithm or another applicable shortest path method, and stored these values in $\delta(v, T)$.

A. The Algorithm

Our algorithm is a dynamically programmed, recursive exploration of all reachable vertex and interval tuples reachable from the root, S. In pseudo-code, our algorithm is as shown in Figure 4.

To construct the BXG for a CFG and budget B, we call bxg(B,S). The BXG is built implicitly in the interval search object; each insert operation adds a vertex to the CFG. Edges are embodied in the constructed hierarchy of INTERVALS. We will refer to a BXG node as $u[i, j]$ where u is the node in the original CFG, and $[i, j]$ is an interval over which all $bxt_x(u)$ are identical, as long as $i \leq x \leq j$.

To extract the BXG, we can walk through the interval search object. For each interval we encounter where the lower limit is not $-\infty$, we emit the node; otherwise, we emit the exception handler[1]. For each child of the node we emit, we create an edge to the copy of the child with an INTERVAL that contains the current INTERVAL, adjusted by the connecting edge weight. Thus, the node $u[8, 9]$ with outgoing edge weight 2 might have children $v1[5, 7]$ and $v2[6, 10]$ (both containing the interval $[8, 9] - 2 = [6, 7]$.)

B. Complexity

There are two types of complexity that matter for this algorithm. First, we have the *computational* complexity of the algorithm. Second, we have the *spatial* complexity of the generated code.

This algorithm is intended for static analysis of program code submitted for admission. The algorithm will run once at admission time and then (if admitted) never again. Thus, while we need the computational complexity to be feasible, we consider spatial complexity to be the more important factor.

[1]Technically, we emit a *call* to the exception handler.

1) Computational Complexity: We can associate each recursive call with an edge in the CFG. Let us examine the algorithm in terms of the number of recursive calls per edge.

For us to insert a vertex-interval pair, we must reach the vertex by a series of bxg calls. Since the remaining cycles R is monotonically non-increasing from B, and we have at most one negatively-lower-bounded interval at each vertex, there are at most $O(B)$ intervals associated with each vertex. We only make recursive bxg calls along outgoing edges on the first failure to find an interval in the interval search object. (Thereafter, the interval will be present.) Therefore, we can make at most $O(B)$ recursive calls along each outgoing edge.

Next, let us examine the number of operations per recursive call. We have two non-constant operations per call—a single search of the interval search object, and a possible single insert into the object. Both can easily be implemented as $O(\log B)$ operations using a standard interval search tree associated with each vertex [3].

By an aggregate analysis over edges, we have $O(mB)$ recursive calls, using m for the number of edges. Each takes $O(\log B)$ time, for a total computational complexity of $O(mB \log B)$.

2) Spatial Complexity: Spatial complexity of the emitted code for a vertex in the CFG depends upon three factors: the number of paths from S to the vertex, the number of paths from the vertex to T, and the budget B.

At each vertex, we emit duplicated code corresponding to each interval that is both *present* and *reachable* from the source S by paths of cost no more than B.

Individual budget values at vertex u are divided into equivalence classes by the weight of each path from u to T. More precisely, we have exactly one interval present for each path from u to T of distinct cost, plus one for exceptions. Therefore, the number of paths of distinct cost from u to T forms an upper bound on the number of intervals that may be present at u.

Each interval corresponds to some number of cycles remaining at this point in the CFG. For an interval to be emitted, it must be *reachable*: there must be a path ρ from S to that vertex such that $B - w(\rho)$ is within the interval. Therefore, the number of paths of distinct cost from S to each vertex is an upper bound on the number of emitted intervals.

As a direct consequence of this and the monotonically non-increasing budget, the number of intervals we emit is upper-bounded by B.

Thus, our spatial complexity is upper bounded by the minimum of three factors: the number of paths to T, the number of paths from S, and the budget B.

C. Natural Extensions

There are two natural extensions of this algorithm that bear mention.

1) Variable Budgets: Our context of network virtualization is event-driven by packet arrival. Performance guarantees are missed in the networking context when packets have arrived at the inputs but are unable to be processed fast enough to forward them to the output at line rate, resulting in output

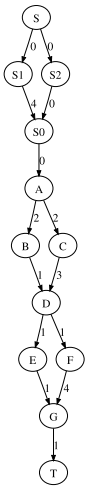

Fig. 5. CFG modified for variable budgets

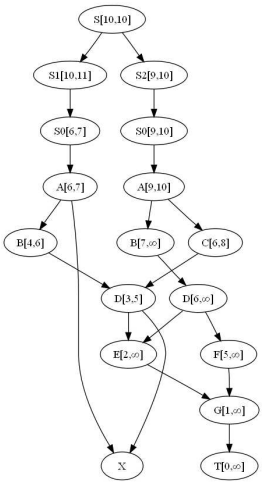

Fig. 6. Bounded execution flow graph (BXG) with variable budgets

underflow and queuing. If the problem is persistent, packets will be lost. In the case of a shared processor, there is no way to guarantee that discarded packets belong to the offending code.

However, not all packets are the same size. Since a larger packet will take more time at the output, we have more time for processing. Fortunately, our model can be easily extended to cope with this situation without changing the algorithm.

Let us take our example of Figure 1 and extend it to handle packets of two sizes, with cycle budgets of 6 and 10 respectively. We can do this by adding code at the beginning of the CFG to check the length of the packet and jump to the appropriate starting point for this length.

See Figure 5. Here we have the modified CFG. Our new start node, S, contains the code to check the packet length and branch to $S1$ for short packets and $S2$ for the long packets. $S1$ and $S2$ do not actually generate code, but are entered into our CFG *as if* they cost 4 and 0 cycles, respectively. To analyze the CFG, we simply call bxt(S,10) as usual, resulting in Figure 6. The algorithm is unaware that no long packets will reach $S1$, but the semantic knowledge is unnecessary. It could serve to reduce the code duplication, of course.

Variable budgets are not free. We do have a small constant cost in the test-and-branch for budget selection. Since we increase the number of early branches, this also serves to drive up duplication of nodes. Nevertheless, for packet processing this is a worthwhile investment.

2) Notify and Continue: We consider it worthwhile in our problem context to consider a modification to the paradigm of partial admission. We currently view the exception handler as an abortion of the code block. However, we could use the exception handler to register a notification that we went over our budget, then continue execution.

This requires a modification to our algorithm. Presently, our algorithm prunes away all subtrees that go over budget. To notify and continue, we would need to return to the flow of execution. To incorporate this notion, we would need to modify the algorithm to add an outgoing edge from the exception handler back to the node we pruned, with an unbounded budget.

Using completely unbounded budgets also requires that our algorithm be adjusted to deal with loops as a special case. Because nodes that sit along paths containing cycles may have an unbounded number of intervals, we would need to explicitly recognize that an unbounded budget forms a special interval, and to handle this separately.

IV. Proofs of Correctness

Our proofs of correctness proceed as follows. First, we present a rigorous treatment of the constructions from CFG to CFT, BXT, and BXG. Next, we prove the key properties of the constructions. Finally, we demonstrate that our dynamic programming algorithm creates our BXG and therefore has all of our required properties.

A. Bounded Execution Subtrees

While it is conceptually clear to proceed from the CFG to the CFT and thence to the BXT, this is mathematically inconvenient. In the case of a cyclic CFG, the depth of the corresponding CFT is unbounded. We prefer to work within the domain of finite graphs. Therefore, we proceed directly from the CFG to the BXT.

Given a CFG $G = (V, E, s, t, w)$ where V is the set of vertices, E is the set of directed edges connecting these vertices, s is our source vertex, t is our sink vertex, and w is a weight function over edges, we construct a BXT T from G as follows.

> Initialize T to have a single vertex r and assign a label $\lambda(r) = s$. For any node u of T, let $p(u)$ be the path from r to u, and extend the weight function w to paths in the natural way. Repeat the following step as long as possible.
>
> Select a leaf u of T with $\lambda(u) \neq t$ and $w(p(u)) \leq B$. Let $v = \lambda(u)$ and let $v_1, \ldots, v_k$ be the successors of v in G. Add nodes $u_1, \ldots, u_k$

to T with edges (u, u_i) and let $\lambda(u_i) = v_i$ and $w(u, u_i) = w(v, v_i)$.

This intermediate construction is the CFT up to and just over the budget frontier. That is, we continue to build on our paths until all leaves u (and *only* leaves) are either over budget or correspond to t. We convert this to our BXT by the following pruning steps.

> For all nodes u in T with path $\rho = r \rightsquigarrow u$ and $w(\rho) > B$, let $\lambda(u) = X$, where X is a new label denoting "exception."
>
> Call a subtree of T open if it contains a node u with $\lambda(u) = t$. Otherwise call it closed. For every node that is the root of a closed subtree and whose parent is not, prune the subtree and let $\lambda(u) = X$.

The tree T obtained in this way is called $bxt_B(s)$. Since the construction can be applied equally well to any node u in G with any non-negative budget B, we can also use $bxt_B(u)$ to refer to any subtree of the BXT rooted at u, so long as we adjust the budget B appropriately.

The $bxt_B(s)$ has four important characteristics: completeness, boundedness, termination, and equivalent functionality.

Theorem 1 (Completeness)**.** If $G = (V, E, s, t, w)$ is a CFG and $T = bxt_B(s)$ is the corresponding BXT, then all paths $s \rightsquigarrow t$ with cost less than B have corresponding paths in T.

Proof: Suppose there were a path ρ in the CFG with $w(\rho) \leq B$ but no corresponding path in T. Without loss of generality, let ρ be the shortest such path, and let $\rho = s \rightsquigarrow u \rightarrow v$. Then $\sigma = s \rightsquigarrow u$ must be in T. However, we have an available construction step from node u, so our construction was incomplete. Thus, T cannot be $bxt_B(s)$. ■

Theorem 2 (Boundedness)**.** If $G = (V, E, s, t, w)$ is a CFG and $T = bxt_B(s)$ is the corresponding BXT, then no path in T has cost greater than B.

Proof: Suppose there were a path ρ in T with $w(\rho) > B$. Without loss of generality, let ρ be the shortest such path, and let $\rho = s \rightsquigarrow u$. If u has no descendants v with $\lambda(v) = t$, then the subtree rooted at u is closed and should have been pruned during the pruning phase. Alternatively, suppose u *does* have descendant v with $\lambda(v) = t$. Because path costs are monotonically non-decreasing, we know that the cost from r to v also exceeds B. Then it should have been relabeled to $\lambda(v) = X$ during the relabeling phase of the pruning step. In either case, T could not have been $bxt_B(s)$. ■

Theorem 3 (Termination)**.** If $G = (V, E, s, t, w)$ is a CFG and $T = bxt_B(s)$ is the corresponding BXT, then all paths in G that exceed the budget B have a corresponding truncated subpath in T terminating at exception node X.

Proof: From the construction, we know that construction continues until for all leaves u either $\lambda(u) = t$ or $p(u) > B$. Thus, for each path in G that exceeds the budget B, there is a corresponding subpath in T that runs beyond the budget frontier.

During the relabeling step of the pruning stage, all nodes u over the budget are relabeled to $\lambda(u) = X$. Since these are all leaf nodes, they represent the roots of closed subtrees. Since the pruning stage can never open a subtree once closed, the corresponding truncated subpath will always terminate in an exception, although it may be further truncated. ■

Theorem 4 (Equivalent Functionality)**.** If $G = (V, E, s, t, w)$ is a CFG and $T = bxt_B(s)$ is the corresponding BXT, then all paths $r \rightsquigarrow u$ in T with $\lambda(u) \neq X$ have labels that correspond directly to paths in G.

Proof: By our construction, no path enters T without coming from a corresponding path in G, and labels are retained pointing back to the original nodes in G. Since the pruning phase only relabels to X, and completes with only leaf nodes relabeled, all labels on safe subpaths in T are retained. ■

We can use T to create a program that is functionally equivalent to the parts of G that stay under budget but which is guaranteed to finish within budget (either at t or at X). For each vertex u in T, we generate code equivalent to $\lambda(u)$ from the original CFG. E.g., see figure 2.

B. Bounded Execution Flow Graphs

The BXT construction repeats many code segment unnecessarily. We can generate a more compact program by merging identical subtrees in T to produce a new CFG, the *Bounded Execution Flow Graph*, $bxg_B(s)$, or BXG.

We first define our notion of equivalent subtrees. Let T be a $bxt_B(s)$ of some CFG. Let there be two nodes u and v in T with children $u_1, \ldots, u_i$ and $v_1, \ldots, v_j$, respectively. We consider the subtrees rooted at u and v to be identical when $\lambda(u) = \lambda(v)$ and all subtrees rooted at corresponding children are also identical.

We begin creating our BXG G from T by copying T completely. Next, we repeat the following step as long as possible.

> Select nodes u and v from G where the subtrees rooted at u and v are equivalent and their parent nodes are not. Merge these subtrees as follows.
>
> Prune v and all descendant nodes from G. For each node we prune, if there is a parent not in the subtree, replace the incoming edge with an edge to the corresponding node in u.

This construction retains all four properties of the BXT, Completeness, Boundedness, Termination, and Equivalent Functionality. Since each property relies upon the (downward) structure of the subtree rooted at each node, and these structures have not changed, no properties have been lost.

In the case of Completeness, no subtrees have been pruned without re-pointing the incoming edges at an equivalent subtree. This applies to all paths, not just paths under budget.

In the case of Boundedness, no paths have been lengthened (or shortened). Thus, if T were properly bounded, so is G.

In the case of Termination, no paths have been lengthened and no nodes have been relabeled.

For Equivalent Functionality, the equivalence property of subtrees depends upon identical labels.

C. Correspondence to Algorithm

Proving that our algorithm corresponds to this construction requires demonstrating several properties of our intervals.

1) Intervals: We prove two properties of intervals to assist in proving that the algorithm corresponds to the BXG construction.

Theorem 5. Given a BXT T generated from CFG G, consider two nodes u and v in T with $\lambda(u) = \lambda(v)$ and identical subtrees. Let $p(u)$ be the path from the root r to u and $p(v)$ be the path from root r to v. Let $i = B - w(p(u))$ and $j = B - w(p(v))$ be the remaining cycles at u and v, and assume without loss of generality that $i \leq j$. If there is a third node z with $\lambda(z) = \lambda(u)$, path $p(z)$ from root r to z, and $i \leq B - w(p(z)) \leq j$ then the subtree rooted at z is also identical.

Proof: Consider the (unbounded) CFT we could generate from any node u in T, consisting of the collective enumerations of the (possibly infinite number of) paths from $\lambda(u)$ to t in G. For each path ρ_i there is a corresponding weight $w(\rho_i)$. This weight does not depend on the incoming budget to u of $B_u = B - w(p(u))$.

We can order these paths as $\rho_1, \ldots, \rho_k$ where $w(\rho_1) \leq \cdots \leq w(\rho_k)$. If we reduce the incoming budget B_u of $bxt_{B_u}(u)$, we will be forced to relabel and prune those leaves where $B_u - w(\rho_i) < 0$. Since our path weights and the ordering are independent of the incoming budget B_u, if we relabel and prune ρ_i then we will also relabel and prune all paths ρ_j with weight $w(\rho_i) \leq w(\rho_j)$.

Now, given u, v known to be the roots of identical subtrees with corresponding cycles remaining $i = B - w(p(u))$ and $j = B - w(p(v))$, $i \leq j$, we know that the leaves of these subtrees have identical labels. Suppose there were some subtree rooted at z with $i \leq B - w(p(z)) \leq j$ and subtree differing from the one rooted at u. Since we know that decreasing the available cycles can never admit additional paths, and $w(p(u)) \geq w(p(z))$, we know that the subtree rooted at u has fewer paths to the sink under budget.

Let us consider these subtrees after relabeling, but before pruning of closed subtrees. To differ from the subtree rooted at u, the subtree rooted at z must have some leaf at the end of a path ρ_k with a label differing from the corresponding leaf in the subtree rooted at u where $B - w(p(z)) - w(\rho_k) \geq 0$ but $B - i - w(\rho_k) < 0$. But because $B - w(p(z)) \leq j$, if $B - w(p(z)) - w(\rho_k) \geq 0$ then $B - j - w(\rho_k) \geq 0$ as well. This implies that $u \neq v$, contradicting our original assumptions. ■

Consequently, for each node u the integers from 0 to B can be divided into subranges such that i and j are in the same subrange if and only if $bxt_i(u) = bxt_j(u)$. We can represent these subranges as intervals $[i, j]$ where $i \leq j$.

For each budget B and node u, there exists a maximal interval $[i, j]$ such that there is no value k not within this interval where $bxt_i(u) = bxt_k(u) = bxt_j(u)$.

Theorem 6. Given a budget $B_u = B - w(p(u))$, a node u with $\lambda(u) \neq t$ and k children $u_1, \ldots, u_k$ and known maximal intervals for each child as $[i_1, j_1], \ldots, [i_k, j_k]$ such that for each xth child, $B - w(u, u_x) \in [i_x, j_x]$, we can compute the corresponding maximal interval for the parent node u as the intersection of the child intervals, each shifted upward by $w(u, u_x)$. That is:

$$[i_u, j_u] = \bigcap_{x=1}^{k} \left([i_x, j_x] + w(u, u_x) \right) \tag{1}$$

is the maximal interval at u containing B_u.

Proof: We use the same construction as in Theorem 5. There are paths from $\lambda(u)$ to t in G. However, these paths consist of the union of all paths from $\lambda(u_x)$ to t in G with $\lambda(u)$ prepended. Let us denote the paths from u_x to our sink as ρ_{xy}. Thus, given the weight of paths ρ_{xy} for u_x as $w(\rho_{xy})$, the weight of the corresponding paths from u are $w(u, u_x) + w(\rho_{xy})$. This accounts for the upward shift by $w(u, u_x)$.

Also as in Theorem 5, these paths can be ordered independently of B_u. Given interval $[i_x, j_x]$ for child u_x, we know that these limits represent the budget points where for some y, $B - w(p(u_x)) - w(\rho_{xy})$ changes sign. (Increasing beyond j_x will cause a negative value $B - w(p(u_x)) - w(\rho_{xy})$ to become zero; decreasing below i_x will cause a positive value $B - w(p(u_x)) - w(\rho_{xy})$ to become negative.)

Since we are given that the upwardly shifted intervals are overlapping, we know that there is some value B_u contained within each shifted interval. That is, $B_u \in [i_x, j_x] + w(u, u_x)$ for all x. If we sort the paths ρ_{xy} in weight order, there will be some smallest value greater than B_u selected from the $j_x + w(u, u_x)$ values, and some greatest value less than B_u selected from the $i_x + w(u, u_x)$ values. These are the endpoints where the most sensitive path changes state. This is the very definition of interval intersection. ■

Finally, by definition, the intervals for node u with $\lambda(u) = t$ are $[-\infty, -1]$ and $[0, \infty]$. That is, we completed under budget if we reached t without going negative on cycles.

Using these maximal intervals, we can merge all subtrees with remaining budgets B within the same interval.

2) Algorithmic Correctness:

Theorem 7 (Algorithm computes maximal intervals)**.** Given a CFG $G = (V, E, s, t, w)$ and budget B, our algorithm generates only maximal intervals.

Proof: Given node u and remaining budget R, our algorithm first looks to see if we already have node $u[i, j]$ such that $R \in [i, j]$. This is our dynamic programming step and exists only for optimization. We may ignore it in our proof.

Next, we check to see if we're over budget (pruning step). Suppose we will exceed our budget. The pre-computed shortest path values allow us to immediately compute the maximal interval without examining the children. First, we know that the subtree rooted at u is closed, as all paths to the sink are longer than our remaining budget R. A closed subtree has an interval unbounded on the left. Finally, we have the shortest

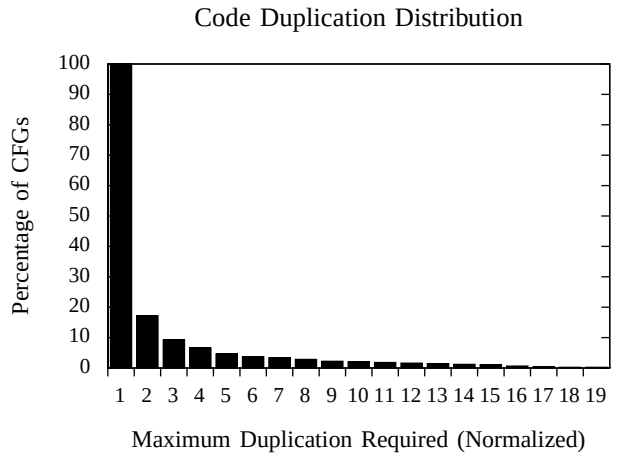

Fig. 7. Percentage of synthetic CFGs requiring more than X duplication (from run of 1000 synthetic CFGs)

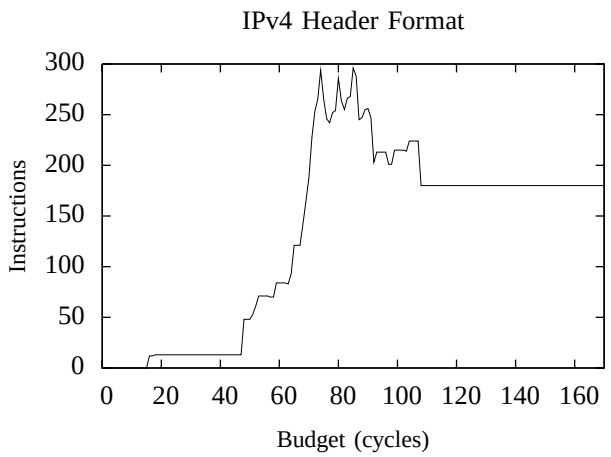

Fig. 8. Code duplication on real CFG (IP Header Format)

path weight $\delta(u, T)$, which tells us at what value of R the first path becomes admissible. This provides the upper bound on the maximal interval.

Suppose we're not over budget. We also check the basis step of our recursive definition in Equation 1. If we match the sink ($\lambda(u) = t$), we can compute the result directly from the basis.

Finally, if we don't have a shortcut, we follow Equation 1.

Since our pruning step computes maximal intervals directly, and our basis step does the same, all we have left is our recursive step. By Theorem 6, this also computes the maximal interval for u.

Therefore, all intervals computed will be maximal. ■

Theorem 8 (Algorithm computes only necessary intervals)**.** Given a CFG $G = (V, E, s, t, w)$ and budget B, our algorithm only computes intervals reachable from the source within our budget.

Proof: Our algorithm proceeds in depth-first search from s, and therefore only visits those vertices reachable from s. Since the cycles remaining is decremented appropriately at each recursive call, we also only investigate those intervals we can actually reach. ■

Theorem 9 (Algorithm Correctness)**.** Given a CFG $G = (V, E, s, t, w)$ and budget B, our algorithm generates $bxg_B(s)$.

Proof: Follows automatically from Theorems 7 and 8. ■

V. Performance

We have implemented this algorithm and tested it on a variety of CFGs and budgets.

A. Synthetic CFGs

Our synthetic CFGs were generated by a series of vertex substitutions that parallel grammar production rules in a C-like language. For our acyclic CFGs, we include simple statements, *if*, *if-then-else*, and *switch/case* statements. For our cyclic CFGs, we added *while*, *do/while*, and *for* loops. In both cases, the typical size of the synthetic input CFG was roughly double the size of the largest packet processing code block we have seen in our router virtualization efforts, and quadruple the target size for a typical code block.

Examine Figure 7. This represents the results of running the algorithm on 1000 different acyclic synthetic CFGs. We show the resulting distribution of the maximum code duplication factor required for each synthetic CFG over all possible budgets. The vast majority (82%) require a maximum duplication factor from 1–2, with an average maximum of 1.6. Large duplication factors are actually very rare; one pathological case required a duplication factor of 23.5. Subsequent analysis of this example showed that it was composed almost exclusively of a series of nested *switch/case* statements.

The results on cyclic CFGs are uninteresting and omitted. While the algorithm works on cyclic CFGs, it works by implicitly unrolling the loop to the limit of the budget. Thus, the code duplication factor is bounded only by the budget. As expected, in simulation the code duplication factor for cyclic graphs is linear in the budget.

B. Real CFG: IPv4 Header Rewriting

For a real CFG, we used the code that rewrites the IPv4 header for next-hop forwarding. This consists of 180 instructions, designed to run at over 5 Gbps on our virtualized router.

See Figure 8. The real CFG necessitated some minor modification to the algorithm to deal with pipeline stalls due to unfilled deferral slots.

At very small budgets, the algorithm actually generates *less* code than the original CFG. This is due to pruning when the budget is too low for this code block. That is, so many paths are pruned that many vertices are never emitted at all. For most application code, this represents a serious developer error and would be reported as such. It is simple for our algorithm to report when certain paths are never admitted, and we implemented this in our experimental version.

Above 108 cycles, we reach the maximum length path of the CFG. At this point, all paths are admissible and no duplication is necessary. The original CFG is accepted with no modification.

A suitable budget for 5 Gbps would be 170 cycles. Clearly, we are under 170. For 10 Gbps we need 85 cycles. The IPv4 header format code is not currently able to achieve 10 Gbps, as the chart makes obvious. Even worse, 85 cycles is the peak of our code duplication, at 296 instructions. This still yields a duplication factor of only 1.64, well in line with our synthetic cases.

VI. Related Work

The major competing technology is WCET analysis using mixed integer programming [4]. This differs from our work in that it makes no effort to solve the code emission problem, and requires that we trust the developer to provide semantic information on branch constraints.

Our problem is different. We need to accept and handle untrusted code in a shared environment. Thus, we must derive any semantic information from the program, not the developer. In the absence of programmer specific semantic information, we can re-write programs to create provably safe CFGs via code duplication.

We also note that the decision to use integer programming to solve the WCET problem was because the developers considered explicit path enumeration infeasible. This fails to consider the possibilities of dynamic programming.

```
for (i=0; i<100; i++) {
  if (rand() > 0.5) j++;
  else k++;
}
```

Fig. 9. "Difficult" WCET analysis for explicit path enumeration

Consider the code snippet in Figure 9. The argument is that this snippet contains 2^{100} possible paths, and that to enumerate them all is simply impractical. However, using a dynamic programming approach with loop bounds, we can determine WCET for this snippet in linear time.

Another approach which bears discussion is *Proof Carrying Code* [5]. In this approach, the developer generates a proof of the correctness of the block of code which can be validated automatically at load time. This approach could be very promising for our problem context. However, it places the burden of generating this proof squarely on the shoulders of the developer. We prefer to allow the developer as much freedom as possible, and generate our own proofs of correctness.

VII. Future Work

Our current implementation of the algorithm does not yet perform emission, nor does it incorporate a parser to accept real-world code. This is our current developmental priority, and requires addressing a number of "real-world" issues we neglect in our theoretical version.

A. Real World Details

First, we have neglected the problem of control flow fallthrough. When a branch is reached, we can re-write the target address. The other side of the branch will simply fall straight through. In consequence, only one block can fall through to another block. To have multiple blocks fall through to the same target requires additional changes.

For cases where paths are not close to budget values, we can simply insert a jump instruction. When this is not practical, we can continue to duplicate code until we reach a point where we can merge the paths. For the IXP architecture, this is likely to be soon—a vacant pipeline deferral slot provides the single slack cycle we need.

Next, the IXP architecture supports asynchronous memory access to allow developers to hide memory latencies. In practical development, both a compute budget and a memory latency budget must be maintained and respected. Adding this functionality to the algorithm appears to be straight-forward, but the impact on code duplication must be examined.

The IXP is a heavily multithreaded environment. In our studies, we have only considered applications with no inter-thread dependencies in packet processing. Higher, trusted layers ensure in-order packet forwarding, but processing code has never required inter-thread dependencies. This assumption is naive and needs to be examined. It may be possible to construct a multi-threaded model of our CFG, analogous to the work with WCET analysis in [6].

Finally, this algorithm only applies to CFGs. A function call has no place in a CFG. Most heavily optimized, high-speed networking code inlines all functions for speed. In these cases, the code represents a CFG. However, for code that does not inline, we have control flow that cannot adequately be represented in a CFG. One approach would be to implicitly inline the function calls and analyze normally; then use a new merge rule to combine inlined function code when possible.

B. Improvements

We have also identified additional ways to reduce duplication. One immediate gain can be made by noting duplicated paths that contain no safe paths "close" to the budget. We can merge these paths by adding runtime checks that lengthen safe paths but do not actually push them over the budget. One possible way to reduce the expense of the runtime check is inspired by Ball and Larus [7], who developed single-counter methods for tracking execution paths through a CFG and applied those to optimize the "hot" paths. In our work, we are interested in using the same techniques to differentiate safe vs. unsafe paths.

Much greater gains can be made by extracting semantic information from the code itself. If we have complete semantic information, we can avoid path enumeration for impossible paths in the CFG. The problem becomes a limited, finite form of the Halting Problem: does this code, when started with any of the possible inputs, halt within B cycles? Any finite form of the Halting Problem is decidable.

We believe that a data flow framework solution is appropriate. With explicit path enumeration, we can solve the constant propagation problem to completion over branch conditions. This would allow us to deduce loop iteration bounds, mutually exclusive paths, and even unreachable code.

We consider this the most important area for additional study. The current state of the algorithm allows duplication to stand in lieu of semantic knowledge. Code that is semantically safe but unsafe in the CFG can be admitted by rewriting the code to guarantee that the unsafe but semantically impossible paths are never taken. With a complete semantic analysis, we would never need to strip those paths, and our code duplication would be reserved for those cases where a genuinely unsafe path is included.

In our application of event-driven, tight budget real-time guarantees, this line of research is very promising. The number of input values to examine is limited by the paucity of available cycles for reading data from memory. We know that our constant propagation will never need to deal with more than a few dozen values, because any code that examines more than this will be over budget due to memory latencies.

VIII. Conclusion

In this paper, we have introduced a new technique for partial program admission. We have demonstrated that dynamic programming can be used to render explicit path enumeration eminently feasible. The same construction can be used to emit a modified CFG that meets event-drive real-time guarantees.

This method shows great promise in the realm of network virtualization. Other applications in similar fields may be equally promising.

References

[1] J. Turner and D. Taylor, "Diversifying the internet," in *IEEE Globecom 2005*, St. Louis, MO, Nov. 2005.

[2] J. Turner and N. McKeown, "Can overlay hosting services make ip ossification irrelevant?" in *Proc. PRESTO: Workshop on Programmable Routers for the Extensible Services of TOmorrow*, May 2007.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. Boston, MA: MIT Press and McGraw-Hill, 2001.

[4] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *SIGPLAN Not.*, vol. 30, no. 11, pp. 88–98, 1995.

[5] G. C. Necula, "Proof-carrying code," in *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, jan 1997, pp. 106–119. [Online]. Available: citeseer.ist.psu.edu/article/necula97proofcarrying.html

[6] P. Crowley and J. Baer, "Worst-case performance estimation for hardware-assisted multi-threaded processors," in *Proc. HPCA-9 Workshop on Network Processors*, 2003.

[7] T. Ball and J. R. Larus, "Efficient path profiling," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 46–57.