Department of Computer Science & Engineering

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

2009-1

Partial Program Admission

Authors: Michael Wilson, Ron Cytron, Jon Turner

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.We present a new technique we call partial program admission, a means of statically enforcing an otherwise untrusted assertion of WCET without adding runtime overhead, by means of code duplication. We apply this technique to real programs from the virtual networking arena and present the results.

Type of Report: Other

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Partial Program Admission

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@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. We present a new technique we call *partial program admission*, a means of statically enforcing an otherwise untrusted assertion of WCET without adding runtime overhead, by means of code duplication. We apply this technique to real programs from the virtual networking arena and present the results.**

I. Introduction

Schedulers for non-preemptive, hard real-time systems require an accurate statement of the worst-case execution time (WCET) of programs. In cooperative environments, we can rely on the developer to provide this information. In less trusted circumstances, we cannot trust the developer to provide accurate WCET.

One important domain that exhibits this problem is high-speed virtual networking. Recent trends in network diversification rely on virtual networks hosted on specialized high-speed network processors [1]. In the case of the Supercharged Planetlab project (SPP) [2], virtual routers run on high-speed network processors such as the IXP 2800 with very high target throughputs. Allowing untrusted virtual routers presents the following challenging requirements: computational budgets are very small and very tight; the developer is not trusted; the IXP microengines do not support timer interrupts.

The SPP platform uses a series of non-preemptive, pipelined microengines running at 1.4 GHz. To reach a bandwidth target of 5 Gbps on minimum-sized packets, a virtual router must process each packet in 185 cycles per pipeline stage. Sharing a processor safely among untrusted virtual routers requires that we bound the WCET of each virtual router to prevent impacting well-behaved virtual routers. Scheduling resources fairly requires that our bound be as tight as possible. The developer may know the true WCET of a virtual router, but we cannot trust the assertion.

What is needed is a method for *enforcing* WCET assertions. In prior work [3] we presented an overview of a method, *Partial Program Admission* (PPA), for bounding execution time by admitting *part* of a program. This works as follows. A developer supplies a program and asserts a putative WCET. We use automated WCET analysis to estimate the WCET.

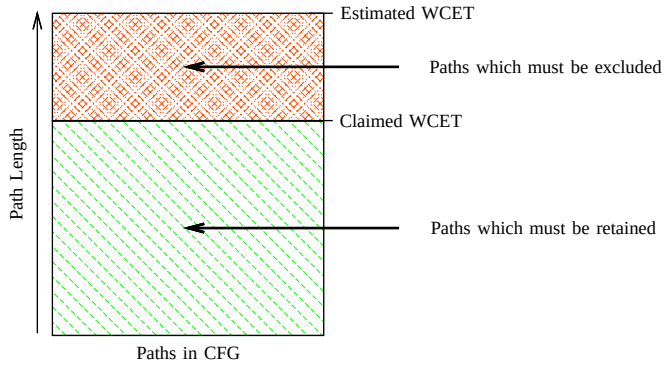

Fig. 1. Partial Program Admission illustrated graphically. Paths within the putative budget are retained. Paths in the CFG which exceed the budget are excluded, whether feasible or infeasible.

As with most automated WCET analysis, we usually overestimate the real WCET. The execution paths of the program for which the execution time exceeds the asserted bound are explicitly rejected by PPA, while those paths under the assertion are retained. We statically re-write the program to retain the admitted paths but exclude the rejected paths.

This can be described graphically, as in Figure 1. Consider a program as a collection of execution paths, given by the structure of a *Control Flow Graph* (CFG). A simplistic way to bound the WCET of a program is to find the longest path through a CFG. However, this path may be infeasible — that is, the logic of the program may render such a path impossible. There is frequently a large gap between an estimate based on the CFG and reality [4]. While much research has gone into computing ever tighter WCET bounds [5], we side-step the problem by re-writing the program to exclude the paths from this middle ground without regard to feasibility.

We emphasize that PPA is a general method, of which we demonstrate a specific example. A developer provides a program and a statement about which execution paths matter ("all paths running in 85 cycles or fewer"). We convert this to a statement about which execution paths must be excluded

("all paths over 85 cycles"), and remove these execution paths.

In our prior work [3], we presented a conceptual overview as well as preliminary results over synthetic CFGs. In this paper, we build on our prior work by presenting an algorithm that implements PPA, along with an evaluation of the algorithm on real applications.

In section II, we define our approach to Partial Program Admission, and present the algorithm and the complexity of the algorithm. In section III, we evaluate our algorithm on a variety of synthetic and real CFGs. In section IV we demonstrate the correctness results. We compare PPA with alternatives in section V, discuss related work in section VI, and future work in section VII. Finally, we summarize our contributions in section VIII.

II. Partial Program Admission

Traditional admission control admits programs that are well-behaved and rejects programs that are not. For our purposes, a well-behaved program is one that always completes within some known time, which we term the *cycle budget*. An ill-behaved program exceeds the budget on some inputs and should be rejected. That is, we can admit a program if the WCET does not exceed the budget, where we define WCET as the longest execution time of the program over all possible inputs.

Unfortunately, perfect admission control requires accurate WCET. In the most general case, determining accurate WCET is exactly equivalent to the Halting Problem and is formally undecidable. Even in realistic scenarios, WCET is often difficult to compute [6].

In PPA, we develop a finer-grained view of a program as a collection of execution paths. We allow the developer to assert a claim of WCET for a program. We can now define well-behaved paths as those which execute within the claimed budget, and ill-behaved paths as those which exceed the claim.

This view of a program allows us to enforce the claimed WCET. When the claim is violated, we interrupt the program and raise an exception. Since we have no preemption, we must raise our exceptions via some static transform of the program. Our application area is high-speed networking, which uses very small budgets, so we cannot afford the penalty of adding runtime checks. Instead, we examine execution paths and intercept ill-behaved paths. Admissible paths are admitted unmodified.

The notion of removing execution paths from a program challenges the usual notion of program correctness. The developer has asserted a WCET bound, or budget. This, combined with the program, is a statement of correctness. From the developer's perspective, a modified program is correct if all paths from the original program not exceeding the budget are present. The developer is responsible for the correctness of the budget. If the developer's assertion is accurate, the program will run exactly as expected. An inaccurate budget is regarded as a developer error justg as any other software defect. From the admission perspective, the program is correct if all paths which would have exceeded the budget are intecepted and an exception is raised. If the developer is wrong or malicious, paths which would overrun the budget are intercepted and an exception is raised. These definitions are not irreconcilable, and we define our correctness as the combination of the two. Functionally, this approach is exactly the same as a timer interrupt approach except that we may raise an exception as soon as we know an overrun will occur; we do not need to wait for the actual overrun.

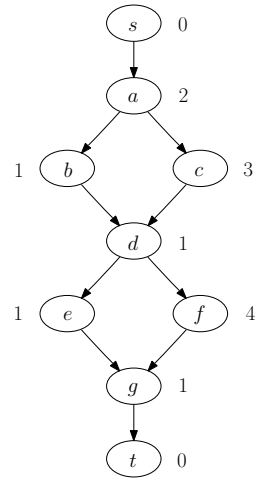

Fig. 2. Control Flow Graph.

Examine Figure 2. We represent our programs by Control Flow Graphs (CFGs) with designated source (s) and sink (t) vertices. Vertex s represents the last trusted code (from a scheduling dispatcher) before beginning the execution of untrusted code (vertices $a - g$). Vertex t represents the point where we resume execution of trusted code, after the untrusted code returns control. All execution within the CFG begins at s and ends at t. We also annotate our graph with vertex weights, representing the cycle cost to execute the code associated with each vertex. Because s and t are trusted code, we account for their execution time separately, and do not charge the untrusted program for their execution costs.

We must observe here that our CFG is for an idealized processor model where execution times are invariant. There is no cache, branch prediction, or instruction pipeline. Therefore, the cycle costs shown in Figure 2 do not depend on any aspect of execution history. This processor model is realistic for a high-speed networking application, with slight modifications to account for an instruction pipeline. (See section III-B.) We also assume that Control Flow Integrity (CFI) holds in this CFG; that is, the program flow follows the CFG. The C compiler for the IXP has restrictions which make CFI tractable to verify.

We expect to receive as input a CFG with weight annotations, special source vertex s and sink vertex t, and putative cycle budget B. The cycle budget is an assertion of WCET by the developer. Since we do not unreservedly trust the developer, we must validate or enforce the budget.

The developer also provides a "time-exceeded" exception

Path	Weight	Result
sabdegt	6	Admissible
sabdfgt	9	Admissible
sacdegt	8	Admissible
sacdfgt	11	Exception

TABLE I
PATHS FROM CFG IN FIGURE 2 AT $B = 10$.

handler, x. x is written to very restrictive standards which allow easy computation of x's WCET. For purposes of PPA, we formally neglect the execution time of the exception handler. In a hard real-time system, we must account for this by suitably adjusting the budget for the possibility of calling x at any time during program execution.

We can view this CFG as representing a set of execution paths, each one a different path through the CFG from source to sink. Our PPA algorithm recognizes those paths which complete under the budget and admits them. Paths which would exceed the budget are rejected. Since these paths represent real code paths, we "reject" them by redirecting to the exception handler. We refer to these as *exception paths*. See Table I.

We generally seek to redirect exception paths at a branch point in the CFG that will definitely cause the budget B to be exceeded. Since exception paths diverge from admissible paths at such branches, we can meet our need by overwriting the destination address of branch instructions with the address of the exception handler. We note that this does not add any time to admissible paths — we only modify exception paths.

Since the affected branch point was itself part of an admissible path, we know that the cost of the path from S to the branch point does not exceed B. Since we formally neglect the execution time of x, no exception path can exceed the budget B.

Our collection of admissible and exception paths can now be viewed as a new CFG. This CFG is guaranteed to meet our cycle budget, because all paths are of length at most B. Unfortunately, some degree of code duplication is necessary to represent our new set of paths. Our PPA algorithm ensures that we have the minimal degree of code duplication required; that is, the new CFG is the smallest CFG that contains precisely the desired paths.

A. Completion Sets

Examining all possible paths through a program is usually infeasible. However, observe that paths often share the same suffix. Once a path suffix has been examined at one budget, any new paths that have the same path suffix may contain overlapping subproblems. This suggests a dynamic programming approach, and our PPA algorithm operates in this way.

A set of path suffixes can be represented by the CFG G and a vertex v. We will refer to this as the completion set of v in G, or $cset(v, G)$, consisting of all paths from v to t within G. We also introduce the notion of the $cset_B(v, G)$, the set of all admissible paths (length B or less) from v to t in G when examined with budget B. Then we can say

$$cset(v, G) = \bigcup_{i \in \mathbb{Z}} cset_i(v, G)$$

Our overlapping subproblem can now be phrased as follows: given CFG G, vertex v, and cycles remaining R, what is the $cset_R(v, G)$?

To answer this, we observe that the completion sets contain some structure. First, observe that as the budget increases, completion sets of the same vertex can only add paths, never lose paths. That is, $cset_R(v, G) \subseteq cset_{R+1}(v, G)$. Second, these path additions take place in discrete steps, whenever R reaches the length of a new path in $cset(v, G)$. Let us define the set $clen_R(v, G)$ as the set of lengths of all paths in the $cset_R(v, G)$, and $clen(v, G)$ analogously as

$$clen(v, G) = \bigcup_{i \in \mathbb{Z}} clen_i(v, G)$$

Note that as R increases, completion sets only change when R reaches a member of the corresponding clen. This allows us to define equivalence classes of cycle budgets.

Let the clen set define a set of *intervals* between adjacent ordered elements. For example, using the CFG in Figure 2,

$$\begin{array}{lcccccc} clen(S, G) = \{ & 6, & 8, & 9, & 11 & \} \\ \text{yields:} & [-\infty, 5] & [6, 7] & [8, 8] & [9, 10] & [11, \infty] \end{array}$$

For convenience, we will treat these intervals as sets of contiguous integers, with the usual set operations. We will also define scalar addition to an interval as shifting the endpoints. That is, $[i, j] + k = [i + k, j + k]$.

All completion sets for budgets within the same interval will be identical. Given the $clen$ sets, we can now answer our overlapping subproblem. If we have already computed a $cset_{R_1}(u, G)$, and we want $cset_{R_2}(u, G)$ where R_1 and R_2 are in the same interval, we have precomputed our answer. Since the completion sets are equal, we can represent these completion sets by the same vertex in our transformed CFG.

Having the complete $clen$ sets for each vertex in the CFG (and therefore the intervals as well) will allow us to compute our transformed CFG directly.

B. Computing the $clen$ Sets

Completion sets of vertices are built directly from the completion sets of successor vertices in the CFG. Given vertex u with successors $v_1, \ldots, v_k$ with known completion sets, we can prepend u onto each path from each $cset(v_i, G)$ to find the paths in $cset(u, G)$.

Since $clen(u, G)$ is just the set of lengths of paths in $cset(u, G)$, we can also compute $clen(u, G)$ from known $clen(v_i, G)$ sets. For each value in each $clen(v_i, G)$ we increment by $w(u)$ and add it to $clen(u, G)$.

Finally, since our sink vertex t has no successors, we have a basis completion set: $cset(t, G) = \{t\}$, and basis $clen(t, G) = \{0\}$. This allows us to compute all $clen_B$ sets as in Figure 3. If we implement the $clen$ sets as bit vectors, the time complexity is $O(mB)$.

initialize $clen(u, G) = \emptyset$ for all u
initialize $pending = [(t, 0)]$
while $pending$ not empty **do**
  Select a pair (v, j) from $pending$ and remove it
  for all edges (u, v) in G **do**
    if $j + w(u) \notin clen(u, G)$ and $j + w(u) \leq B$ **then**
      Add $j + w(u)$ to $clen(u, G)$
      Add $(u, j + w(u))$ to $pending$
    end if
  end for
end while

Fig. 3. Computing $clen$ sets.

C. Computing the Transformed CFG

The $clen$ sets define all of the intervals for each vertex. The $clen_B$ sets are equivalent so long as we never consider R values that exceed B. We can define a new graph, D, which tracks the remaining cycles R at each vertex by the interval into which each R would fall.

We are interested in the equivalence classes of vertices with identical completion sets. For this purpose, we will name our vertices in D by a 2-tuple of the vertex and interval, where the interval contains R. Notationally, we will refer to $u[I]$, or when using the interval endpoints, $u[i, j]$.

Given $G = (V, E)$, let us define $D = (V', E')$ as follows.

$$\begin{aligned} V' &= \{u[I] \mid u \in V, I \text{ defined by } clen(u, G)\} \\ E' &= \{(u[I], v[J]) \mid (u, v) \in E \text{ and } I \subseteq J + w(u)\} \end{aligned}$$

If we apply this to the CFG of Figure 2, we get the graph in Figure 4. This is our dynamic programming graph, which we can use to determine R-bounded completion set equivalence.

Observe the following key properties, for which we defer proofs to Section IV. First, paths in D correspond directly to paths in G. Second, given a value R and vertex $u[I]$ such that $R \in I$, those paths from $u[I]$ ending at $t[0, \infty]$ have length not exceeding R, while those paths ending at $t[-\infty, -1]$ have length exceeding R. That is, those ending at $t[0, \infty]$ correspond to members of $cset_R(u, G)$. This allows us to immediately determine when all paths from a vertex exceed the budget, yielding our exception paths. That is, for any value R in the interval of an exception vertex $u[I]$, $cset_R(u, G) = \emptyset$. Finally, given $u \in V, R_1, R_2$ in the same interval, then $cset_{R_1}(u, G) = cset_{R_2}(u, G)$, and inversely.

Recall that our goal is to produce a new CFG G' from G and B where all paths of length B or less are included; all paths of length exceeding B are intercepted. That is, we want a new CFG where $cset_B(s, G)$ is admitted and all other paths in $cset(s, G)$ raise exceptions.

Let H be the interval at s which contains B. From the properties of D, we know that all paths from $s[H]$ to $t[0, \infty]$ have length B or less; all paths from $s[H]$ to $t[-\infty, -1]$ have length exceeding B. Further, we can recognize the exact branch at which a path is certain to exceed B.

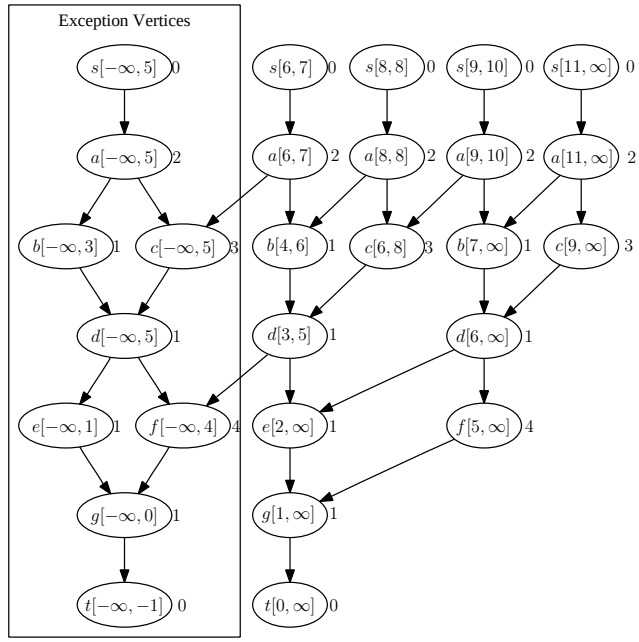

Fig. 4. Dynamic programming graph D generated from Figure 2. Exception vertices are those vertex/interval pairs with empty completion sets.

Since our goal is to generate a program which raises an exception as early as possible, and which returns control to the trusted code immediately after, we must modify our graph D. Let D' be a new graph where all exception vertices are coalesced into a single vertex, x, and add an edge from x to $t[0, \infty]$. See Figure 5.

We can now generate our transformed CFG G' as simply those vertices of D' reachable from $s[H]$. One algorithm to do so in $O(m') = O(mB)$ is in Figure 6. For this, we require a vector for each vertex u which records the interval defined by $clen(u, G)$ corresponding to a given cycle count $R < B$. Using this, we can find the interval J that contains $I - w(u)$ in constant time. We can compute this vector in $O(nB) = O(mB)$ time using the values of $clen(u, G)$.

Following our running example, G' is shown in Figure 7.

An alternative variation to compute G' directly from G can be implemented by recursive descent augmented by memoization, taking the same worst-case time complexity. As expected, the sparse memoization yields a speed improvement.

D. 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. The generated code will be installed and take execution store for the lifetime of the deployment. Thus, while we need the

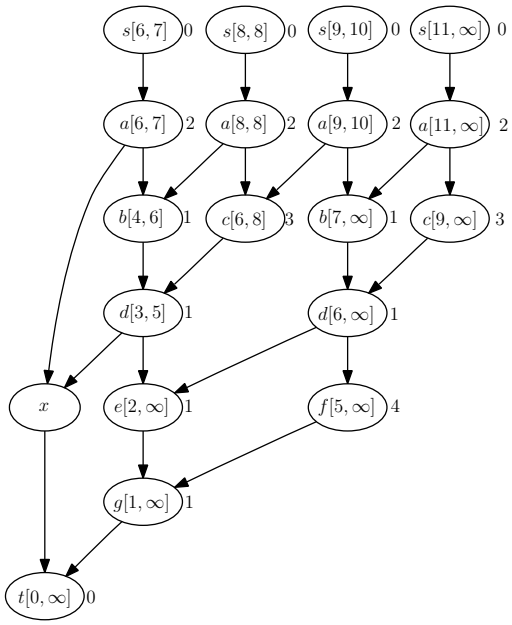

Fig. 5. Dynamic programming graph used by algorithm of Fig. 6 on Fig. 2. x takes the place of vertices associated with lower-unbounded intervals, where the corresponding completion set is empty. For example, $cset_4(c, G) = \emptyset$ and so $c[-\infty, 5]$ is replaced by x. The edge $x \rightarrow t[0, \infty]$ is included to return control to trusted code after an exception.

given $D' = (V, E), S[H]$
initialize $V' = \{S[H], X, T[0, \infty]\}$
initialize $E' = \{(X, T[0, \infty])\}$
initialize $pending = [S[H]]$
while $pending$ not empty **do**
  Select a vertex $u[I]$ from $pending$ and remove it
  for all edges (u, v) in G **do**
    let J be the interval of $clen(v, G)$ containing $I - w(u)$
    if $v[J]$ is an exception vertex **then**
      Add $(u[I], X)$ to E'
    else if $v[J] \notin V'$ **then**
      Add $v[J]$ to V'
      Add $v[J]$ to $pending$
      Add $(u[I], v[J])$ to E'
    end if
  end for
end while

Fig. 6. Computing CFG G'.

computational complexity to be tractable, we consider spatial complexity to be the more important factor.

We have already addressed computational complexity, which is $O(mB)$.

The code is emitted directly from G', so the spatial complexity of the emitted code is proportional to the complexity of the vertex set of G'.

Vertices $u[I]$ in G' each correspond to a vertex u in G. We will refer to these as copies of u and consider the number of copies of each vertex in G which can exist in G'.

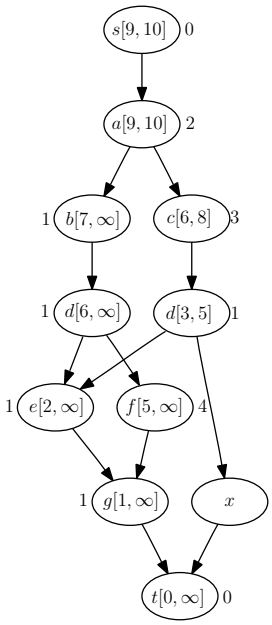

Fig. 7. Bounded execution graph G' from Figure 2 at $B = 10$.

Spatial complexity of the copies of u depends upon three factors: the number of paths from s to u, the number of paths from u to t, and the budget B.

At each vertex in G, we add a copy to G' 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 in $cset(u, t)$. More precisely, we have exactly one interval 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 copies of u and is exponential in the branching factor of G.

Each interval corresponds to some numbers of cycles remaining at this point in the CFG. For a copy $u[I]$ to be added to G', the associated interval I must be *reachable*: there must be a path ρ from s to u such that $B - w(\rho)$ is within the interval I. Therefore, the number of paths of distinct cost from s to u is an upper bound on the number of copies of u, and is also exponential in the branching factor of G.

Since vertices in G' are of the form $u[I]$, intervals have minimum size of 1, and we only need consider intervals below the budget B, the number of copies per vertex is upper-bounded by B.

Thus, our spatial complexity per vertex 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. That is,

$$\#\text{copies of vertex } u = O\big(\min\big(\#\text{paths } s \rightsquigarrow u, \#\text{paths } u \rightsquigarrow t, B\big)\big)$$

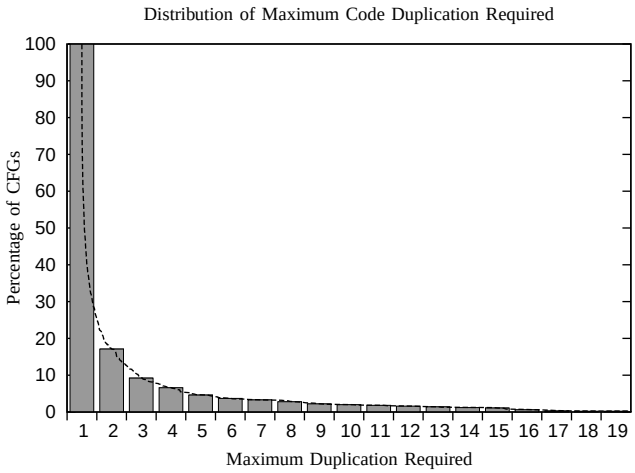

Fig. 8. Percentage of synthetic CFGs requiring more than X duplication (from run of 1000 synthetic CFGs).

III. Evaluation

We have evaluated the algorithm on a series of synthetic CFGs and CFGs from real code. The results on synthetic CFGs have been previously published in [3]; we briefly recapitulate these results here. We also describe here our examination of the algorithm on real CFGs from a network processing environment.

A. Results on Synthetic CFGs

Our synthetic CFGs were generated by a series of vertex substitutions that parallel grammar production rules in a C-like language. These CFGs were restricted to acyclic cases and included simple statements, *if*, *if-then-else*, and *switch/case* statements. The mean size of the CFGs was 3600 instructions. This is more than double the largest real program we studied, and provides a useful bound on performance on larger programs.

Figure 8 represents the results of running the algorithm on 1000 randomly generated 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.

Results on cyclic synthetic CFGs are not meaningful in the absence of a WCET assertion. As demonstrated in subsection II-D, the code duplication is linear in the budget, and our synthetic evaluations confirmed this. Without a known WCET of interest, the maximum duplication for cyclic CFGs cannot be evaluated.

B. Adaptation to Real Code

Our algorithm has been designed for an idealized processor model which lacks a number of real-world details. In our analysis on real code, we adapted the base algorithm to run on assembly code for the IXP 2800 architecture.

The IXP 2800 architecture uses a 4-stage pipeline. The pipeline requires that we account for pipeline aborts on branches. This can be done in a straightforward manner, by inserting dummy vertices into the CFG. These dummy vertices carry no actual code, but have a cost equal to the pipeline abort costs of the branch. We inserted one of these dummy vertices into each flow transfer edge with a pipeline abort cost.

The IXP 2800 supports up to 8 hardware thread contexts per processor. Context switching fragments CFGs and leads to greater difficulties in WCET analysis [7]. In this evaluation, we ignore the problem of inter-thread dependencies. Our CFGs are examined under a strictly single-threaded model.

The IXP 2800 also has a memory hierarchy of local registers, very fast local memory, off-chip SRAM banks and off-chip DRAM banks. There is no automated data or instruction cache; all memory management is manual. Asynchronous memory access instructions allow for masking of high memory latencies. These asynchronous memory access instructions typically allow a program to start a memory read, perform some processing, and then context-switch to await the results. We neglect "swapped out" memory latency in our evaluation, although we consider this an important detail for future work.

We must also ensure that our CFG is correct, that CFI holds for our analysis. Fortunately, the C compiler and architecture for the IXP have some fundamental limitations that make this relatively easy. First, there is no stack. Return addresses are stored directly in registers. In consequence, there is also no recursion. We can ensure that returns from functions are sane just by verifying that the return register is untouched. Second, there are no function pointers, so all function calls are explicit and immediate. If we do not allow inline assembly, then CFI holds automatically. To allow inline assembly, we can simply interdict constructions which are difficult to prove correct.

C. Sample Code

Our sample code comes from high-speed network processor modules written for the Supercharged Planetlab (SPP) project [2] and plugins for the Open Network Laboratory (ONL) [8]. See Table II.

Our list of programs is as follows. `Count` is a simple packet counter. `Nstats` gathers statistics on the proportion of different protocols within the packet stream. `Port_count` and `port_reporter` are a matched pair; `port_count` tracks the TCP and UDP ports seen in the packet stream, and `port_reporter` reports this information to a centralized store. These programs were all student-written.

`Ipv4_parse` parses IPv4 headers, which may be encapsulated in tunnel headers, and performs RFC 1812 router verifications. `Ipv4_hdrfmt_encap`[1] rewrites IPv4 headers for next hop forwarding, including encapsulation within UDP tunnels. `I3_parse` and `i3_hdrfmt_encap` perform

[1]We manually analyzed a similarly-named program in [3]. `Ipv4_hdrfmt_encap` is distinct and does significantly more work.

Program	Size (Instructions)	Cyclic?
`count`	23	No
`nstats`	368	No
`ipv4_parse`	614	No
`ipv4_hdrfmt_encap`	497	No
`i3_parse`	734	No
`i3_hdrfmt_encap`	582	No
`port_count`	229	Yes
`port_reporter`	133	Yes

TABLE II
SAMPLE PROGRAMS FOR EVALUATION.

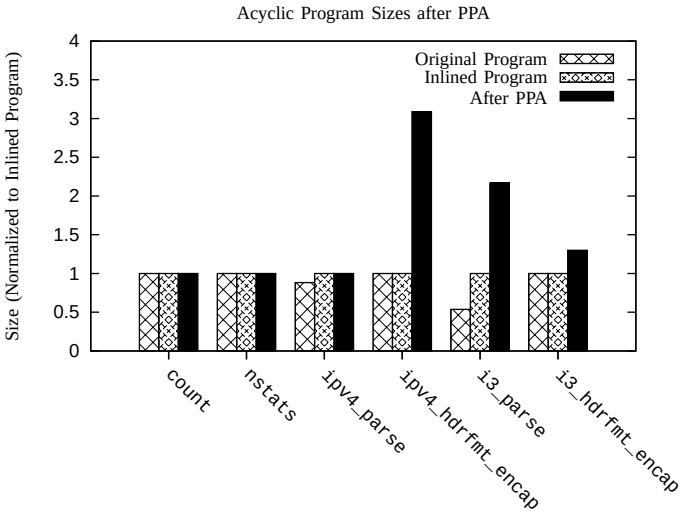

Fig. 9. The results of running our PPA algorithm on a series of programs at their respective real WCETs. All programs were fully inlined before analysis. All programs were acyclic.

analogous tasks for the *Internet Indirection Protocol* (i3) [9], a novel architecture for using indirection as a means of giving users greater control over the traffic they receive. These programs were written as part of the SPP project and were developed to rigorous standards of performance. Therefore, most function calls were inlined, and all loops were unrolled for speed.

To examine this code with our algorithm, we needed to make a number of changes to the programs. First, our algorithm only works on CFGs that do not contain function calls. Therefore, all function calls were inlined. Second, some code contained inter-thread dependencies. We have changed these programs to conform to a single-threaded model. Third, our initial examination showed that cyclic code resulted in substantial duplication, so we have analyzed the cyclic code in both the original state and after unrolling all loops to produce acyclic code. Finally, we have removed debug code which would not be present in a production system.

We divide our presentation as follows. First, we present the results on acyclic programs. We follow this with the cyclic programs, first analyzed as cyclic programs, then analyzed after unrolling all loops. Finally, we take a detailed look at the results on one program over a range of budgets.

1) Acyclic Programs: Figure 9 shows the results of the analysis on acyclic programs. All sizes have been normalized to the inlined program, including the original program size.

We manually analyzed these programs for real WCET, and asserted this as a budget in our PPA algorithm. For the first three programs, the WCET was the length of the longest path in the CFG. In these cases, the program could be admitted *in toto*, with no modification at all.

For the next three programs, the real WCET was shorter than the longest program path. In these cases, some duplication was necessary to differentiate admissible and exception paths.

The `ipv4_hdrfmt_encap` analysis brings out an interesting point. We define the WCET as the largest execution time over all possible inputs. However, we have information about the inputs that is not available in either the CFG or the `ipv4_hdrfmt_encap` program. The compiler generates memory accesses without assumptions about memory alignment. In this case, we know that the inputs will be such that memory accesses will be properly aligned. Therefore, despite the fact that these paths are feasible over all inputs, we do not regard these inputs as possible. This reduces the WCET to 192 cycles. Even a perfect WCET analysis tool would be unaware of this domain knowledge. Working solely from the `ipv4_hdrfmt_encap` program, the correct WCET would be 203. In this respect, our PPA has capabilities not provided by perfect WCET-based admission control.

2) Cyclic Programs: Two programs contained loops. These were short, static iteration count (4) loops used to traverse small tables.

Pure tree-based WCET analysis is unable to bound cyclic code at all, as no iteration bounds are available. A common solution to the problem is to provide language constructs for static iteration bounds on loops, whereupon the analysis proceeds by assuming maximum iterations [4]. There are also approaches to automatically bounding loop iterations [10]. In our analysis, we do not provide any iteration bounds — the number of iterations is ultimately bounded only by the cycle budget.

This has the effect of implicitly unrolling each loop to the limit of the budget, resulting in significant code duplication. Nevertheless, we can still produce bounded programs even in the absence of any iteration bounds at all. See Figure 10.

In the case of the `port_reporter` program, the duplication factor is substantial, 13.98. However, even at this factor, the transformed program is only 3,048 instructions. This readily fits into the IXP 2800 instruction store (8,192 instructions).

The duplication on the `port_count` program is more reasonable at less than 6. The duplication factor here is due mostly to structural characteristics of the program rather than the budget. In particular, a loop does not prohibitively increase program size if the rest of the program path lengths are in a small range, and the WCET assertion is accurate.

3) Unrolled Programs: Despite the fact that our PPA algorithm works on unbounded loops, we examined the results of unrolling the loops to decrease the duplication. This brought

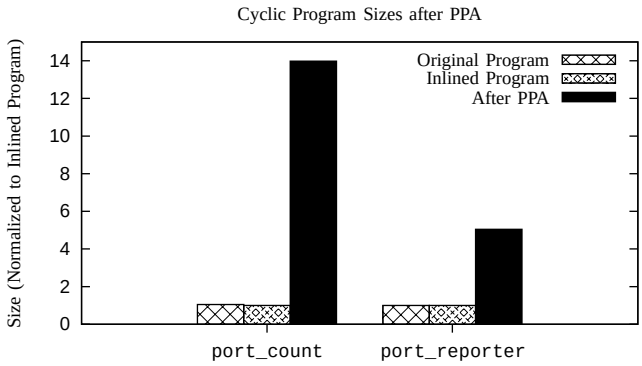

Fig. 10. The results of running our PPA algorithm on a pair of cyclic programs at their respective real WCETs. All programs were fully inlined before analysis.

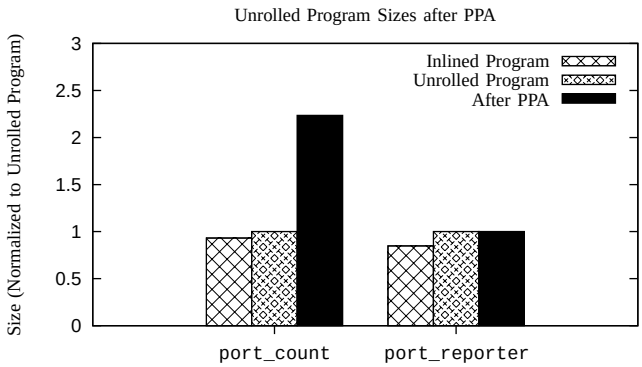

Fig. 11. The results of running our PPA algorithm on a pair of unrolled cyclic programs at their respective real WCETs. All programs were fully inlined before unrolling, and fully unrolled to an acyclic configuration before analysis.

the results back into line with the rest of the acyclic programs.

See Figure 11. After unrolling, the `port_reporter` program's WCET was equal to the length of the longest path in the CFG, resulting in admission without duplication. For `port_count`, some duplication was still necessary.

Our goal is to share a processor between bounded-execution programs. It is a significant result that we can fit all 8 transformed acyclic programs (7,018 total instructions) into the same IXP 2800 instruction store (8,192 capacity) with more than enough room remaining for a scheduler.

D. Analysis with Range of Budgets

When the WCET is equal to the length of the longest path in the CFG, we admit the program *in toto*, with no path pruning and no consequent duplication. When the WCET is shorter, we have some degree of duplication. We now explore the degree of possible duplication over a range of budgets for the `ipv4_parse` program.

Observe Figure 12. This is a fairly typical duplication curve for an acyclic program. The longest program path was 414 cycles, and at this budget or higher, no duplication

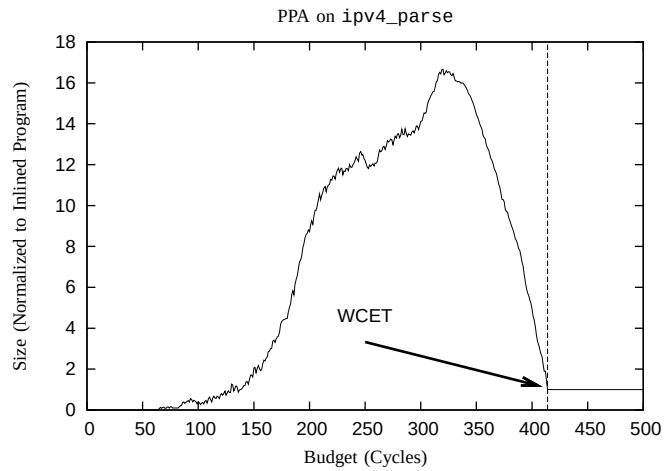

Fig. 12. Code duplication of `ipv4_parse` on various budgets. The real WCET (414) is equal to the longest path, so there is no code duplication at this budget.

arises. Below this value, path pruning requires duplication to differentiate cycles remaining to successor vertices. Our worst possible duplication factor, 16.66, occurs at 320 cycles.

Below 190 cycles, we begin to prune all paths containing certain vertices. We no longer admit any copies of these vertices. Below 137 cycles, we have pruned so many paths that the resulting program is smaller than the original, and at 64 cycles, all paths are rejected.

For completeness, we also present the duplication curve on a cyclic program, `port_count`. See Figure 13.

In this case, there is no longest path, and the duplication factor continues to grow linearly in the budget. Asserting an accurate WCET becomes a crucial part of mitigating the duplication factor. In this case, the real WCET is 268, for a duplication factor of 13.98.

The same discussion at low budgets applies to cyclic programs. Below 123 cycles, we begin to prune all copies of some vertices; that is, there are no admissible paths which include these vertices. At 112 cycles, we emit a smaller program than the original, and at 33 cycles no paths are admitted.

IV. Correctness

Our demonstrations proceed as follows. First, we re-state the properties of the graph D from section II-C in mathematical terms. These are proven as lemmas. We follow this with the important results: the new program is *complete*, in that all paths which the developer requested are present; the program is *bounded*, in that all paths are of length at most B. These two results taken together meet the correctness definitions we gave in section II. Finally, we close with a proof that the new program is as *compact* as possible.

Lemma 1. *Given $G = (V, E), u, v \in V, (u, v) \in E, I$ defined by $clen(u, G)$, then there is exactly one interval J defined by $clen(v, G)$ that intersects $I - w(u)$.*

Proof:

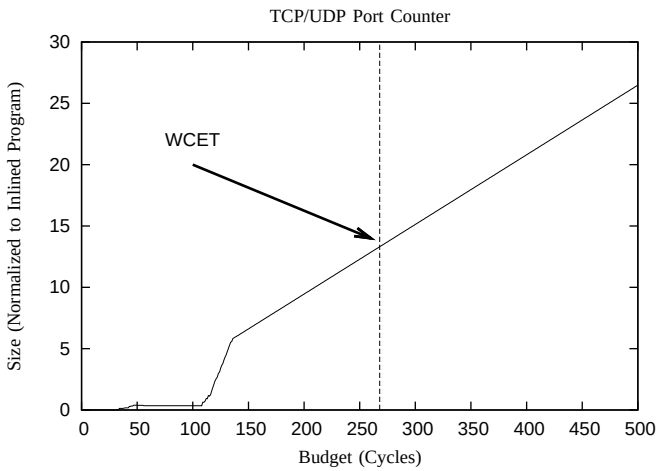

Fig. 13. Code duplication of TCP/UDP Port Counter program on various budgets. The real WCET is 268, bounding our final duplication at 13.98

By definition, intervals at the same vertex v cover the entire set of integers. Also by definition, intervals are non-empty. Therefore, there must exist at least one interval $J = [j, k]$ defined by $clen(v, G)$ such that J intersects any other interval, including $I - w(u)$. Without loss of generality, let J be the largest such interval (greatest lower bound j) at v.

Let $i = min(I)$.

If $j > i - w(u)$, then there exists a path $u \rightsquigarrow t$ in G of length $j + w(u)$, which implies that $clen(u, G)$ defines an interval lower-bounded by $j + w(u)$. But because intervals at the same vertex are disjoint, this would imply that J cannot intersect $I - w(u)$. Therefore, $j \leq i - w(u)$.

The $clen(v, G)$ can define no other $J' = [j', k'] \neq J$ which also intersects $I - w(u)$, because $j' < j \leq i - w(u)$. ■

Corollary. *Given the same situation as in Lemma 1, $I - w(u) \subseteq J$.*

Proof:

There is exactly one intersection, J. If $I - w(u) \not\subseteq J$, then some other interval at v would intersect the remainder of $I - w(u)$. ■

Next we demonstrate that $cset_i(u, G)$ "matches" the paths in $u[I] \rightsquigarrow t[0, \infty]$ in D. By *matches* we mean that if we strip the intervals from the vertices in the paths in D, the sets of paths are identical. This requires a series of lemmas.

We introduce a new notation, $\delta(u, v)$ to be the shortest distance from vertex u to vertex v in G. Distance refers to the sum of the weights of the vertices in the path.

Lemma 2. *Given $G = (V, E)$ and $D = (V', E')$ with $u[I] \in V'$ and $i = \delta(u, t) \in I$, and let $P(u[I])$ be the set of paths $u[I] \rightsquigarrow t[0, \infty]$ in D shorn of intervals, then $cset_i(u, G) \subseteq P(u[I])$.*

Proof:

We proceed by induction on path lengths.

Basis. Path length of one, $u = t$. Then immediately from the definitions, $i = \delta(t, t) = 0$, $I = [0, \infty]$, $P(t[0, \infty]) = \{t\} = cset_0(t, G)$.

Inductive Step. Given path $\rho \in cset_i(u, G)$ where $i = \delta(u, t)$, let the second vertex of ρ be v. This implies that $(u, v) \in E$, and $\delta(v, t) = i - w(u)$. Let ρ' be the path when u is removed from ρ. Then $\rho' \in cset_{i-w(u)}(v, G)$. Let our inductive hypothesis be that $\rho' \in P(v[J])$. Then by the definition of D, edge $(u[I], v[J]) \in E'$, since J intersects $I - w(u)$. Therefore, $\rho \in P(u[I])$ as well. ■

Lemma 3. *Given $G = (V, E)$ and $D = (V', E')$ with $u[i, j] \in V'$, and let $P(u[i, j])$ be the set of paths $u[i, j] \rightsquigarrow t[0, \infty]$ in D shorn of intervals, then $cset_i(u, G) \subseteq P(u[i, j])$.*

Proof:

First, note that for $i < \delta(u, t)$ this is trivially true, as $cset_i(u, G) = \emptyset$. We now consider $i \geq \delta(u, t)$, and we demonstrate by induction on i.

Basis. $i = \delta(u, t)$. This is exactly Lemma 2.

Inductive step. Let ρ be a path in $cset_i(u, G)$, and let v be the second vertex along ρ. (If there is no second vertex, then $u = t$, $\rho = t$ which matches the path $t[0, \infty]$ in D for all $i \geq 0$.) Then let ρ' be the path when u is removed from ρ. Then $\rho' \in cset_{i-w(u)}(v, G)$. Let our inductive hypothesis be that $\rho' \in P(v[J])$. Then by the definition of D, edge $(u[I], v[J]) \in E'$, since J intersects $I - w(u)$. Therefore, $\rho \in P(u[I])$ as well. ■

We can now state our completeness theorem, which proves that the program meets the developer's perspective of correctness.

Theorem 1 (Completeness)**.** All execution paths that do not exceed B are present in the transformed CFG G'.

Proof:

By definition, all paths in G' are in $P(s[H])$ if $B \in H$, so we can state this theorem mathematically as $cset_B(s, G) \subseteq P(s[H])$. This is immediate from Lemma 3. G' is actually generated from D' (see Section II-C), which is a modified version of D. However, $P(s[H])$ in D' is a superset of $P(s[H])$ in D, as we have only added the exceptions paths. ■

We can also demonstrate the converse by a similar set of lemmas. That is, no paths have been incorporated into D or D' (and therefore G') except for those of $cset_B(s, G)$ and the exception paths. We omit these proofs, as we can resort to an argument that the vertices of CFG G' have meaning. Any path arbitrarily added to G' would be unreachable code; there would be no flow transfer to match the edge by which the extra path was added.

Theorem 2 (Boundedness)**.** All paths in the transformed CFG G' have lengths less than B.

Proof:

All exception paths diverge from admissible paths. The exception handler is treated as having no cost. Thus, the exception path length is not longer than at least one admissible path, with length B or less. ■

Theorem 3 (Compactness)**.** There is no other graph meeting Thms 1,2 with smaller size than G'. That is, G' is (one of) the most compact program(s) meeting our requirements.

Proof:

This proof maps the CFG G' onto a Deterministic Finite Automaton (DFA) accepting strings isomorphic to paths in the completion set, then applies the Hopcruft-Ullman algorithm [11] to demonstrate that G' is maximally compact.

When we speak of program size, we refer to the amount of code. Hopcruft-Ullman counts the number of states. Therefore, to apply the algorithm directly, we must break our vertices in G into single-instruction vertices. For purposes of our PPA algorithm this is purely cosmetic.

Given $G = (V, E), D' = (V', E')$, and G' defined as that portion of D' reachable from $S[H]$ where $B \in H$, we can construct DFA $M = (Q, \Sigma, \delta, q_0, F)$ as follows. (Note that we now deal with D' rather than D.)

The states Q of M are taken directly from the vertices V' of D, with a additional trap state q_T for undefined transitions and one explicit accept state, q_F.

The alphabet Σ is taken from the edges E of G.

Our transition function δ is defined as follows.

$$\delta(u[I], (u, v)) = \begin{cases} v[J] & \text{If } (u[I], v[J]) \in E' \\ x & \text{If } (u[I], v[J]) \notin E' \text{ and} \\ & \quad (u[I], x) \in E' \\ q_F & \text{If } u[I] = t[0, \infty] \text{ and} \\ & \quad (u, v) = (t[0, \infty], q_F) \\ q_T & \text{Otherwise.} \end{cases}$$

q_0, our start state, is defined as the root of G', $S[H]$.

Our set of accepting states $F = \{q_F\}$.

This DFA M accepts sequences of edges. It is easy to show by induction that the paths as sequences of edges are isomorphic to paths as sequences of vertices. Following the properties of D', we know that M accepts $cset_B(s, G)$, with the rest of the paths (if any) in $cset(s, G)$ replaced with exception paths.

We can now directly apply the Hopcruft-Ullman algorithm. This consists of two phases: first, we remove all vertices not reachable from the start state. This is exactly our construction of G' from D'.

Second, we partition the states of the DFA into equivalence classes based on strings accepted. Two states are equivalent if, were the DFA started in these states, they each accept exactly the same set of strings. By our construction of D', we know that each state $u[i, j]$ accepts strings corresponding to paths in $cset_i(u, G)$ and the related exception paths. These are already in equivalence classes via intervals; each state in M will accept a different set of strings.

Finally, we construct the new, minimized DFA using the equivalence classes as states in the new DFA. Since we have exactly one state in each equivalence class, the minimized DFA is isomorphic to G'.

Therefore, G' is minimal. ■

V. Alternatives

PPA is one of several approaches to solving timer overrun problems, each of which has different trade-offs. In general, there are 4 approaches to the problem: timer interrupts, instrumentation of programs with runtime checks, WCET analysis with whole program admission, and PPA. We will compare these approaches, with recommendations on when to use each approach.

Timer interrupts are the most common approach. The scheduler sets a hardware timer and calls the untrusted code. If the code completes within the alotted time, the timer is reset and the next program can be called. Otherwise, a hardware interrupt returns control to the scheduler. This requires that the hardware support low overhead timer interrupts, but has the advantage that we do not need any low-level architectural knowledge at all. When available, this is usually the best approach.

Instrumenting programs with runtime checks of time remaining is straightforward and requires minimal understanding of program logic or underlying architecture. Unfortunately, it adds some additional cycles to the execution time. It also adds a (small) amount of code. This approach is best when WCET analysis is exceptionally difficult and some overhead can be tolerated.

WCET analysis uses automated analysis of a program to estimate the WCET. Having this estimate, we can directly determine whether to admit or deny the program. This approach requires that our estimates are reasonably tight, or admission control becomes overly pessimistic. Obtaining tight estimates of WCET usually requires substantial time and architectural information, and implementations are often difficult. We need to solve the dual problem of single-instruction WCET analysis, which depends upon the underlying architecture, and arbitrary program behavior. This approach is usually best when budgets are loose and some pessimism in admission control will not impact the application greatly.

PPA works on a path-by-path basis to support developer assertions of WCET. This places a burden on the developer to estimate the WCET, although there is no requirement that the estimate be tight. It also requires underlying architectural information to solve the single-instruction WCET analysis, but avoids the need to understand program behavior. It has the advantages of adding no execution time, but the disadvantage of increasing program size. This is best when budgets are small, reducing duplication, and tight, where runtime checks or traditional WCET analysis may be inappropriate.

VI. Related Work

We have already discussed other approaches to the same problem of bounding execution time. Here, we discuss research that uses similar underlying techniques.

Duplicating paths to exploit different characteristics along the path history is not new. In [12], paths are duplicated so that data flow problems, unsolvable along one path, can be solved in the duplicate. This allows superior compiler optimizations

Method	Use when
Timer	Timers are available and have low overhead
Runtime Checks	Overhead is acceptable
WCET	Hardware model is tractable; and program behavior can be fully understood or budgets are loose
PPA	Budgets are small and tight, hardware model is tractable

TABLE III
COMPARISON OF APPROACHES TO TIMER OVERRUN.

along heavily used paths while bypassing infrequent paths which cannot be so optimized.

PPA is closely related to, but distinct from, WCET analysis. We only estimate WCET of paths until we find that a path will exceed the budget, and intercept that path. In this way, we sidestep the problem of path feasibility which WCET must address. However, since we rely on our estimate to exclude paths, the per-path estimate must be tight or we may exclude safe paths in violation of the developer's correctness requirements. In traditional WCET analysis, overestimating a single path can only lead to overestimation of the total program WCET.

PPA is also distinct in that even a perfect WCET analysis would not fill all of the uses of our PPA technique. Consider the case of `ipv4_hdrfmt_encap`, where the developer has access to preconditions which might not be available to a WCET analysis. These preconditions, on memory alignment, render the longest path infeasible. Unless a way exists to express these preconditions to the WCET analysis tool, we will always overestimate WCET. PPA allows the developer to assert the WCET directly.

Distinctions aside, PPA is founded on simplistic WCET analysis techniques, and improvements to the underlying estimation may also improve PPA performance. WCET analysis falls mostly into two distinct methods, tree-based path enumeration [4] and implicit path enumeration [13]. In tree-based path enumeration, depth-first examination of the CFG combined with a pessimistic "worst-of-successor" tally of instruction times yields a fast estimate of WCET. In the purest form, no effort is made to avoid counting infeasible paths, so this method typically overestimates WCET significantly. Extensions to limit counting of infeasible paths [14] typically increase analysis time unacceptably. This renders most tree-based techniques unsuitable for admission control. While our analysis portion is most closely related to the tree-based techniques, we use the information to reject paths, not programs. Therefore, we do not reject complete programs because one long path cannot be shown to be infeasible.

Implicit path enumeration uses information from the developer on branch constraints to develop a much tighter estimate of WCET. We regard the developer as untrusted, and cannot rely on developer assertions for safety. Implicit path enumeration is most useful in PPA when a developer uses it to determine real WCET, prior to program submission.

Transforming programs to make demonstrations of WCET easier are also not new. Closest in spirit and functionality is the Single-Path approach to writing temporally predictable code [15]. The single-path approach is a variant of WCET analysis where we remove data-dependent branches from a program to yield a program with one execution path. Once the program is reduced to a single path, WCET analysis is trivial.

Data-dependent branches are removed by executing *both* sides of the branch and retaining only the correct result using predicated instructions. Loops are retained but converted from data-dependent iteration counts to fixed iteration counts. The results of unnecessary loop iterations are discarded. Both transformations have the net result of increasing WCET by some amount.

In contrast, our technique increases code size by duplicating partial paths as a means of retaining path history, while WCET is untouched.

We see these techniques as complementary, applicable to different arenas. Given a very small cycle budget, the overhead of any added runtime is unacceptable. The single-path approach would penalize programs significantly. Since our code duplication is bounded by the budget, this environment also keeps the duplication factor small.

Given larger budgets, the situation is reversed. Code duplication can balloon, while the addition of a few cycles of runtime in the single-path approach is a minor cost.

We believe that a synthesis of the two techniques could be interesting. Our code duplication is bounded by branching factors which the single-path approach can limit; the WCET penalty of the single-path approach could be mitigated by some code duplication.

VII. FUTURE WORK

Our priority in future work is techniques to reduce the duplication factor, especially for cyclic programs. The algorithm as presented here substitutes code duplication for knowledge of path feasibility. We believe that we can incorporate feasibility information in two ways.

First, we can incorporate feasibility information from a trusted source by representing infeasible paths as regular expressions within the CFG in a manner similar to the work in [14].

Second, we can incorporate *untrusted* feasibility information represented in the same way, by taking developer assertions as statements of correctness.

Recall that the general PPA method involves transforming a program to remove interdicted paths, while retaining paths which the developer requires. In current work, all paths are strictly classified into one category or the other.

Labeling infeasible paths allows a middle ground, where some paths are acceptable but not required. This allows flexibility in the transformation which could be used to further reduce code duplication. Provably infeasible paths can be retained or omitted without impacting safety. Assertions from the developer may not be trusted, but amount to a statement from the developer that "I consider this path optional. Keep

it or throw it away as you please." This introduces the notion of optional paths, consisting of the provably infeasible paths and the admissible developer-flagged paths. The new problem becomes one of retaining the optional paths which result in the smallest transformed program.

These techniques can apply to both acyclic and cyclic code, although our main interest is in cycles. In the case of a cycle, techniques such as in [10] could provide trusted bounds on loop iterations. Where these techniques fail, the developer can still assert, "This loop iterates 3 times; I don't care if an exception is raised on a fourth iteration because it won't happen."

A similar difficulty arises in applying PPA to programs which contain function calls. A function call has no place within a CFG. In our current work, we have inlined all function calls, as is common practice in packet processing code when optimizing heavily for speed. However, we believe that this difficulty can also be overcome by extending our base PPA algorithm to use infeasible path labels to assert valid return addresses.

Another difficulty arises when considering code emission. The bounded CFGs we generate cannot be directly emitted as executables. Our method occasionally yields CFGs where *multiple* copies of a vertex "fall through" to the same *single* copy of a subsequent vertex. Since this is not possible in real executables, additional research is necessary to convert to an executable form. We believe that a combination of techniques can surmount this difficulty. Limited additional duplication can solve most multiple fall-through cases. In other cases, we expect to be able to prove that the insertion of an explicit branch instruction will not violate our bounds.

Our current implementation does not perform code emission from bounded CFGs. We plan to develop a complete system as a proof of concept, from real code to executable code.

In our real system, we expect to examine other timing factors beyond computation. The IXP 2800 processor supports primitives for asynchronous memory I/O, along with up to 8 hardware thread contexts with single-cycle context switches. Our programs usually have both tight computational cycle budgets as well as larger, coarse-granularity, memory latency budgets. In practice, our system will need to demonstrate the ability to bound memory latency budgets as well as computational budgets.

VIII. Conclusion

We have demonstrated a new technique for admission control of untrusted programs, Partial Program Admission. The "all or nothing" limitation of traditional admission control requires that we prove that an entire program is safe before admission, even when the "unsafe" parts correspond to infeasible execution paths. Under PPA, we can admit just those portions of the program which are safe, and exclude portions of the program where safety proofs are impractical.

We have also demonstrated an implementation of a PPA algorithm for bounding the execution time of untrusted programs, and have proven its utility on real programs from the high-speed networking context. We also enforced our budgets with no runtime overhead.

Finally, we have demonstrated two additional advantages of PPA over traditional admission control based on WCET analysis. First, no knowledge beyond the control flow structure is needed. Second, we can bound programs at budgets below the analytical WCET, such as when input restrictions render some paths infeasible.

References

[1] 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.

[2] J. S. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar, "Supercharging Planetlab: a high performance, multi-application, overlay network platform," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 85–96, 2007.

[3] M. Wilson, R. Cytron, and J. Turner, "Partial program admission by path enumeration," in *Proceedings of the Work-In-Progress Session of the 14th Real-Time and Embedded Technology and Applications Symposium*, 2008, pp. 97–100.

[4] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Journal of Real-Time Systems*, vol. 1, no. 2, pp. 159–176, Sep. 1989.

[5] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[6] R. Kirner and P. Puschner, "Obstacles in worst-cases execution time analysis," in *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Orlando, Florida, May 2008.

[7] P. Crowley and J. Baer, "Worst-case performance estimation for hardware-assisted multi-threaded processors," in *Proc. HPCA-9 Workshop on Network Processors*, 2003, pp. 36–47.

[8] J. DeHart, F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong, "The open network laboratory," in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2006, pp. 107–111.

[9] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *In Proceedings of ACM SIGCOMM*, 2002, pp. 73–86.

[10] C. Healy, M. Sjdin, V. Rustagi, and D. Whalley, "Bounding loop iterations for timing analysis," in *In Proceedings of the IEEE Real-Time Applications Symposium. IEEE CS Press, Los Alamitos, Calif*, 1998, pp. 12–21.

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[12] G. Ammons and J. R. Larus, "Improving data-flow analysis with path profiles," 1998, pp. 72–84.

[13] 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.

[14] C. Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Syst.*, vol. 5, no. 1, pp. 31–62, 1993.

[15] P. Puschner and A. Burns, "Writing temporally predictable code," in *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 85.