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 knowlege 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.

II. Algorithm Overview

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

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.

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.

A. 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 edges are labeled by the execution time required for the correspondig 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

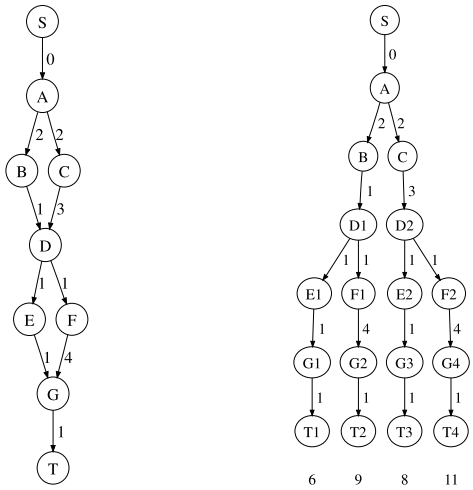

Fig. 1. CFG and the corresponding CFT. Weights along the edges represent cycle counts to traverse that edge. Total path cycle counts are presented below each terminal node in the execution tree.

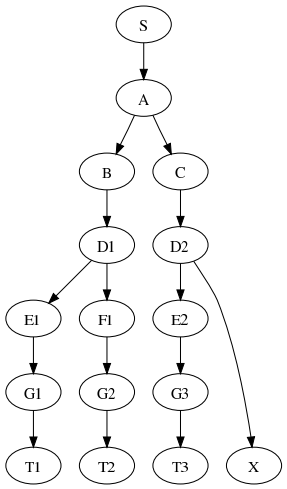

Fig. 2. Abort to exception handler

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.

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.

B. Code Duplication Reduction

The BXT typically contains multiple 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 (that is, they represent copies of the same orignal 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 coalesced 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 coalesce 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 coalesce 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.

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 that u_1 was derived from and the amount of available execution time that remains after execution has reached u_1. If the length 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

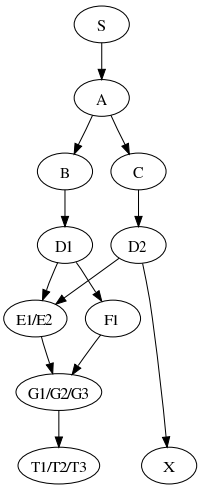

Fig. 3. Coalescence of equivalent execution subtrees

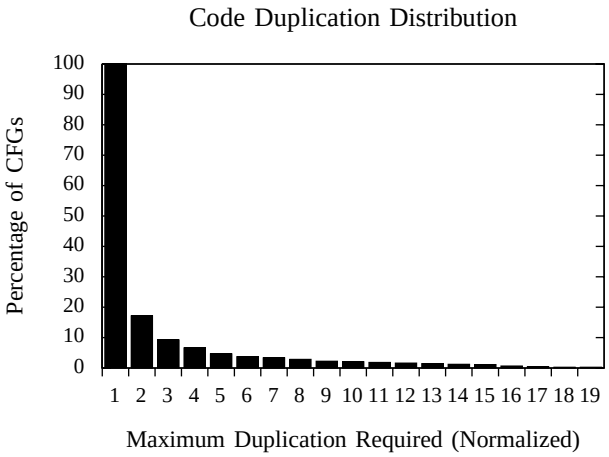

Fig. 4. Percentage of synthetic CFGs requiring more than X duplication (from run of 1000 synthetic CFGs)

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 lengths of their paths from the root are identical. More generally, if their path lengths are p and q, 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 0 to B. Two values i and j fall in the same block 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. See [3] for a complete description of the algorithm, a correctness proof and execution time analysis.

III. 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 4. 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.

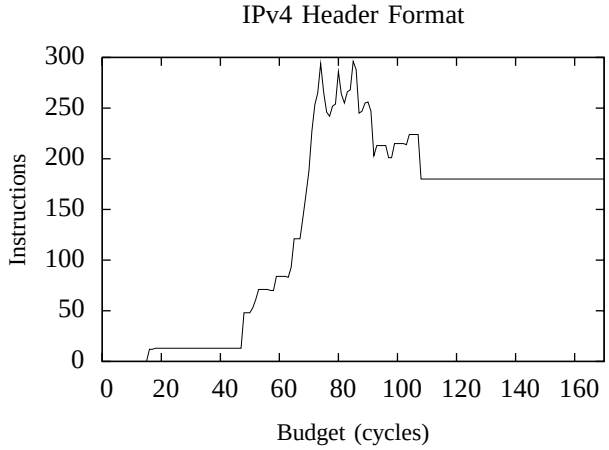

Fig. 5. Code duplication on real CFG (IP Header Format)

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 actual 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 5. 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.

IV. 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. 6. "Difficult" WCET analysis for explicit path enumeration

Consider the code snippet in Figure 6. 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.

V. Continuing 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.

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 coalesce 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 [5], 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 decideable.

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.

VI. 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] M. Wilson, R. Cytron, and J. Turner, "Partial program admission by path enumeration," Washington University, St. Louis, MO, WUCSE Tech. Rep. WUCSE-2008-4, 2008.

[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] 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.