Trickle: Automated Infeasible Path Detection Using All Minimal Unsatisfiable Subsets

Bernard Blackham†, Mark Liffiton‡ and Gernot Heiser†

†*NICTA and University of New South Wales, Sydney, Australia*
‡*Illinois Wesleyan University, Bloomington IL 61701, USA*
Email: Bernard.Blackham@nicta.com.au, mliffito@iwu.edu, gernot@nicta.com.au

***Abstract*—Static analysis techniques can be used to compute safe bounds on the worst-case execution time (WCET) of programs. For large programs, abstractions are often required to curb computational complexity. These abstractions may introduce infeasible paths which result in significant overestimation. These paths can be eliminated by adding additional constraints to the static analysis. Such constraints can be found manually but this is labour-intensive and error-prone. Automated methods of finding infeasible path constraints are thus highly desirable.**

In this paper we present Trickle: a method to automatically detect infeasible paths on compiled binary programs, in order to refine WCET estimates. We build upon the Sequoll framework and apply satisfiability modulo theory (SMT) solvers to find classes of infeasible paths. Unlike other techniques, Trickle can find infeasible paths which contain an arbitrary number of conflicting conditions. We also integrate the compute all minimal unsatisfiable subsets (CAMUS) algorithm to reduce the number of refinement iterations required. We show the practicality of Trickle by applying it to a WCET analysis of the seL4 microkernel. We also evaluate its effectiveness on the Mälardalen WCET benchmarks.

***Keywords*-Real time systems; Operating system kernels; Software verification and validation;**

I. Introduction

Infeasible paths in worst-case execution time (WCET) analyses are a source of overestimation and thus inaccuracy. They arise when WCET analyses utilize an abstraction of a program to hide (mostly) irrelevant details, in order to curb the computational complexity of the analysis. Such analyses may consider more program paths than are actually possible. Although this ensures that the results given by the analysis are sound (i.e. all possible paths are considered), some of these paths cannot be executed in practice. Infeasible paths can be eliminated by refining the program abstraction, leading to more accurate WCET estimates, and thereby reducing the amount of overprovisioning required for real-time systems.

The problem of infeasible path detection arises in a variety of applications, including program compilation and bug detection. In this paper, we look at the problem in relation to a specific method of WCET analysis—the implicit path enumeration technique (IPET). IPET [1], described further in Section III-A, constructs an abstraction of the possible paths through a program using linear equations. Without further constraints on the equations, the abstraction created by IPET can be overly coarse, as it does not consider the values of variables and the satisfiability of conditional expressions. As a result, many infeasible paths may be created.

To refine the abstraction created by IPET, additional constraints can be introduced into the system of equations. These constraints can be broadly classified into two categories: those that rely on knowledge of global invariants, and those which can be deduced through local reasoning. The first category can generally be identified quickly by a developer with an understanding of the code base, but is significantly more difficult to automate. This is because automatically deducing global invariants is computationally expensive—although some techniques exist to "guess" possible program invariants [2]. It is much easier to automatically detect infeasible paths which fall into the second category (i.e. due to local reasoning). It is these paths which we focus on in this paper.

In this paper, we present Trickle—a method to automatically refine the abstraction used by IPET for computing WCET. We build upon Sequoll [3] and its ability to reason about compiled binaries. We utilise an SMT solver and integrate the *CAMUS* algorithm [4] for identifying unsatisfiable subsets within a system of constraints. We apply Trickle to the WCET analysis of the seL4 microkernel [5], and also evaluate it on the Mälardalen WCET benchmarks [6].

In previous work applying the Sequoll framework, infeasible paths were tediously specified manually by a human. Trickle automates the process of eliminating infeasible paths, speeding up computation of the worst-case execution time, while avoiding the potential for human error created by manual annotations. Trickle has three key advantages over past approaches: (1) unlike many traditional abstraction-refinement techniques, this method is well-suited to IPET-based WCET analyses, as the structure of the infeasible paths found can be expressed precisely as integer linear equations; (2) this method can detect infeasible paths arising from the interaction of an arbitrary number of mutually unsatisfiable conditions, whereas some techniques are limited to only pairwise conflicts; and (3) by using an SMT solver in a directed iterative refinement, our method is more precise than traditional interval-based value analyses.

```
int f(int a) {
    if (a & 4)
        slow();
    else
        fast();
    if (!(a & 4))
        slow();
    else
        fast();
}
```

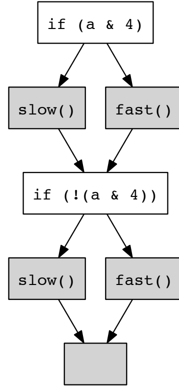


Figure 1: A program with a typical "double diamond" control flow graph and bit-arithmetic. There are two infeasible paths in this program as the if statements are mutually exclusive.

```
int f(int a, b) {
    if (a == 0)
        ...
    if (a == b)
        ...
    if (b == 1)
        ...
}
```

Figure 2: A program containing a "3-diamond." Any two conditionals can be simultaneously satisfied, but all three cannot.

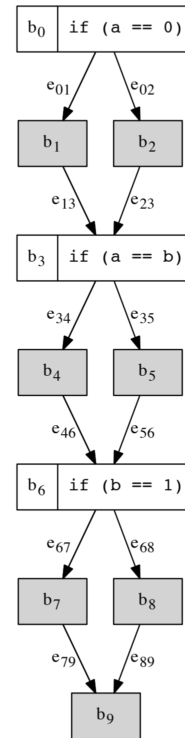


Figure 3: Control flow graph corresponding to the program in Figure 2, showing variables for the execution counts of each basic block and edge.

II. Motivating examples

We focus on finding infeasible paths in order to refine the WCET estimates of a program using IPET. There has been a significant amount of past research on finding infeasible paths, especially for WCET analysis (described further in Section VI). Many algorithms are framed in the context of a common construction known colloquially as the "double diamond." Such a case arises from two mutually exclusive conditional statements, such as the example shown in Figure 1. This case is complicated by infeasible paths derived from bit arithmetic, common in embedded systems but is not handled by existing infeasible path detection schemes suitable for IPET.

As a second example, consider the program in Figure 2 with the control flow graph shown in Figure 3. No pair of conditionals in this program are mutually exclusive, however all three conditionals cannot be simultaneously satisfied. One distinct advantage of the Trickle approach is that it generalises naturally to infeasible paths involving an arbitrary number of conflicting edges, such as those arising from n-diamonds. An infeasible n-diamond configuration consists of n conditionals such that any strict subset may be simultaneously satisfiable, but the set of all n conditionals are not.

III. Background

This section summarises the relevant background behind the implicit path enumeration technique used to compute worst-case execution time and introduces the CAMUS algorithm developed by Liffiton & Sakallah [4].

A. WCET computation by IPET

There are many possible approaches to compute the worst-case execution time of a program, each with their own advantages and drawbacks. Wilhelm et al. have published a review of the state of the art in worst-case execution analysis tools and techniques to which we refer the reader [7]. Computing the precise WCET of large programs executing on modern hardware architectures requires using abstractions of the program and/or the hardware. The choice of abstraction selects a compromise between accuracy and scalability. Here, we only focus on refining the program abstraction, not the hardware.

We use the implicit path enumeration technique pioneered by Li et al. [1] and implemented in Chronos [8]. IPET computes the WCET by viewing it as a linear optimization problem. It formulates a set of integer linear equations where variables are used to represent the execution counts of each basic blocks, as well as each edge between blocks. The flow constraints from the control flow graph of the program are encoded as linear equations in an integer linear programming problem, as are the constraints on loop bounds. The overall execution time of the program is the objective function whose maximum value is found by an integer linear programming (ILP) solver. The objective function is given as the sum of products of every basic block's execution count and execution time.

In its primitive construction, the equations encode only the structure of the control flow graph and ignore the flow of any data through the program. It is entirely possible for a worst-case path found in this construction to be infeasible in practice, for example, because of conditional branches that cannot be satisfied. If the constraints which cause a path to be infeasible can be expressed as a linear equation, then the equation can be added to the ILP problem to exclude it. Note that not all infeasible paths can be expressed this way—for example, any constraint involving non-linear arithmetic cannot.

In the remainder of this paper, we use b_i both to refer to the basic block and as a variable denoting the execution count when part of an equation. Similarly e_{ij} refers to the edge from b_i to b_j, as well the variable denoting the number of times that edge is taken.

Given our example in Figure 3, we can express the fact that all three conditionals are mutually unsatisfiable by the following equation:

$$e_{01} + e_{34} + e_{67} \leq 2$$

This equation states that the sum of execution counts of the three "true" edges in the control flow graph (CFG) is at most two, therefore no path will traverse all three edges. Note that this expression is only correct if this fragment of the control flow graph only executes once (i.e. if it is not part of a loop). We address loops in Section IV.

B. *Computing all minimal unsatisfiable subsets*

Satisfiability modulo theories (SMT) solvers can be used to determine if a given set of constraints is satisfiable. Modern SMT solvers can reason about constraints expressed as formulae over Booleans, integers, bitvectors, lists and more. If the constraints are not satisfiable, many SMT solvers simply return the unenlightening result "unsatisfiable". However, some will endeavour to provide a smaller unsatisfiable subset or "core" of the original problem to explain why the problem is unsatisfiable, weeding out irrelevant information.

In the context of worst-case execution time, every edge of a loop-free control flow graph can be viewed as a Boolean formula of the conditions that must hold for the edge to be traversed. Given a set of edge conditions along some execution path, an SMT solver can determine whether the edge conditions can be simultaneously satisfied. If not, that path is infeasible, and an unsatisfiable subset can be used to form an additional constraint that eliminates an entire class of infeasible paths including the one tested.

A *minimal unsatisfiable subset* (MUS) is an unsatisfiable subset of constraints of which removing any individual constraint allows the others to be satisfied. A set of formulae may contain multiple MUSes, which themselves may be disjoint or overlap. The problem of finding MUSes has seen an increased interest recently due to its increasing importance in formal verification [9]. Two approaches have been proposed for finding *all* minimal unsatisfiable subsets [4], [10].

In this paper, we apply the *compute all minimal unsatisfiable subsets* (CAMUS) algorithm developed by Liffiton & Sakallah [4]. The full details of the algorithm are given in their paper, however we will present a brief overview here to give an understanding of the limitations when applied to WCET analysis.

MUSes are closely related to the concept of *minimal correction subsets* (MCSes). An MCS is a minimal subset of constraints in an infeasible constraint system such that when the MCS is removed, the remaining constraints are feasible. Each MCS is the set-wise complement of a *maximal satisfiable subset* (MSS)—a satisfiable subset of the constraints which cannot be expanded any further without making it unsatisfiable.

Let Ω be a collection of sets of elements from some domain $\mathcal{D}$. $H \subseteq \mathcal{D}$ is a *hitting set* of Ω iff every set within Ω is "hit" by an element of H—i.e. each set of Ω shares a common element with H. An *irreducible* hitting set of Ω is a hitting set where no element can be removed without losing the hitting set property. For example, let $\Omega = \{\{A, D\}, \{B, C, D\}\}$ (and our domain $\mathcal{D} = \{A, B, C, D\}$). Then $H_1 = \{A, B, C\}$ and $H_2 = \{A, B\}$ are both hitting sets of Ω but only H_2 is irreducible.

A duality exists between the MUSes and MCSes of an infeasible constraint system: every MUS is an irreducible hitting set of the set of all MCSes. Similarly, every MCS is an irreducible hitting set of the set of all MUSes. The CAMUS algorithm leverages this duality to compute all MUSes, by first computing the set of all MCSes (as the MCSes are easier to compute). Only once all MCSes are found can the MUSes be computed.

This method of construction gives rise to some challenges in managing the computational complexity when computing MUSes. In particular, given a set of *disjoint* MUSes, the number of MCSes is the combined product of the cardinalities of each MUS—i.e. given the disjoint MUSes $\{M_1, M_2, \ldots, M_n\}$, the number of MCSes is given by $|M_1| \times |M_2| \times \ldots \times |M_n|$. This is a worst case for

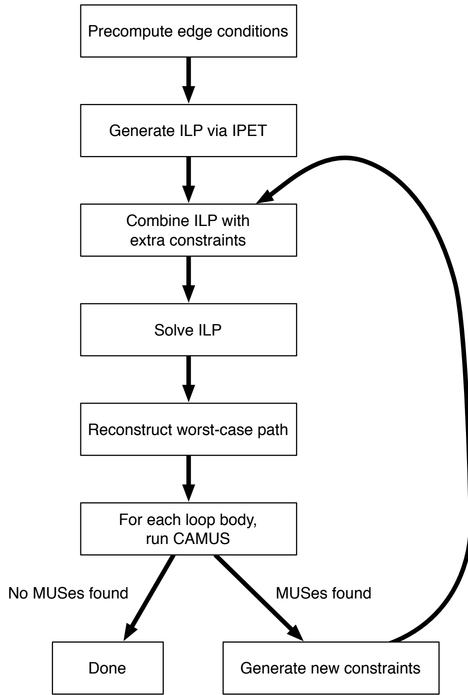


Figure 4: An outline of the Trickle algorithm

the CAMUS algorithm as the number of MCSes grows exponentially with the number of MUSes.

The CAMUS algorithm builds upon an existing constraint solver. It searches for all maximal satisfiable subsets, giving all minimal correction sets as their complement, which in turn are used to compute all minimal unsatisfiable subsets as their hitting sets. The CAMUS algorithm has been applied to both Boolean satisfiability (SAT) and SMT solvers. Our work builds upon the SMT implementation of CAMUS using the Yices SMT solver [11]. As such, our implementation is limited to loop bounds based on conditionals which can be encoded and decided by the Yices SMT solver and the theories it can support.

IV. Details

The outline of our algorithm is shown in Figure 4. Using the Sequoll framework, we first precompute the *edge conditions* of every edge in the control flow graph. An edge condition is a Boolean expression corresponding to a condition which must hold in order for that edge to be traversed. For example, the edge condition on e_{34} from Figure 3 is $a = b$, and similarly for e_{35} is $a \neq b$. As Sequoll has transformed our program into single static assignment (SSA) form [12], all edge conditions relate to SSA variables, making them natural to express to an SMT solver.

We note that using an SMT solver is significantly more powerful than past approaches using interval-based analyses, as SMT solvers can evaluate more complex constraints—e.g. constraints involving bitfields, which are not uncommon in embedded software.

Using standard IPET techniques, we generate a set of integer linear equations which encode the structure of the control flow graph. This is generated using the Chronos tool [8]. As Chronos also models the instruction and data caches, the resulting ILP equations are more complex, but they have the same inherent structure as produced by the standard IPET approach.

By solving the ILP equations, we obtain an assignment of values to our basic block and edge count variables. From these values, we can reconstruct the path through the abstract program using an Eulerian path algorithm. At least one Eulerian path is guaranteed to exist due to the flow constraints for each node—every path into a node must exit, except for the global source and sink where the path originates and terminates, respectively.

Given the longest path through the control flow graph, we can collect all edge conditions of all edges on this path. This gives us our set of constraints which we can test for satisfiability with an SMT solver. If the constraints cannot be satisfied, then we invoke the CAMUS algorithm, which returns a set of MUSes corresponding to all sets of edge conditions that are mutually unsatisfiable. Each such set describes a class of paths which are infeasible.

We note that the CAMUS algorithm may return no MUSes even when the SMT problem is not satisfiable. This can arise if the constraints are too complex causing the SMT solver to time out, or in some circumstances (under some logical domains) the SMT solver is unable to decide if the constraints are satisfiable or not.

If no MUSes are returned (because either the constraints are satisfiable, or CAMUS returns no MUSes) infeasible paths may still exist in the control flow graph. Loops (discussed below), non-linear arithmetic, unresolvable memory accesses, and invariants on the code which are not expressed to the SMT solver, can all create infeasible paths which cannot be found using SMT with Trickle. When no further MUSes are found, we terminate and return the most recent worst-case path.

On the other hand, if the CAMUS algorithm returns sets of infeasible constraints, we convert each set (MUS) into an equation and augment the existing ILP in order to refine our abstraction. For example, consider the program listed in Figure 2, with its control flow graph shown in Figure 3. Assume that the worst-case path detected so far included the edges e_{01}, e_{34} and e_{67} (and possibly others beyond the graph shown). The CAMUS algorithm would detect that these edges are not satisfiable and return the single MUS $\{e_{01}, e_{34}, e_{46}\}$. This can be converted into a constraint of

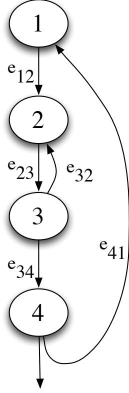


Figure 5: An example CFG with nested loops

the form:

$$e_{01} + e_{34} + e_{67} \leq 2$$

In general, any such MUS on a loop-free control flow graph $M = \{e_1, e_2, \ldots, e_n\}$ can be converted into an ILP constraint of the form:

$$e_1 + e_2 + \cdots + e_n \leq |M| - 1$$

We repeat the process again using the original set of ILP equations, augmented with any constraints derived from MUSes, until no further MUSes are found. By eliminating the class of all paths containing any MUS found, the ILP solver will not find them on subsequent iterations, guaranteeing progress.

Now we have seen the general idea behind the algorithm, there are some details to be addressed.

Loops: The approach used by Trickle is intrinsically tied to loop-free control flow graphs. The reason for this limitation is two-fold: first, SMT solvers cannot natively reason about loops except by explicitly unrolling them, or by incorporating loop invariants [13]; second, even if we could find infeasible paths across loop iterations, it may not be possible to construct a corresponding ILP constraint. There are some limited classes of constraints which can be expressed [14], but in the general case it is difficult, if not impossible.

However, we can detect infeasible paths if they lie within a single iteration of a loop. Our approach considers each loop within a program separately. For each loop L (a strongly connected component in the CFG), we select the edges that are contained in L, but are not a part of any nested loops within L. In the example CFG shown in Figure 5, there are two loops, one nested within the other. We consider the set of edges of the outer loop $\{e_{12}, e_{34}, e_{41}\}$, and the set of edges of the inner loop $\{e_{23}, e_{32}\}$. We do not consider irreducible loops [15] in our analysis (i.e. loops with multiple entry points), which means that the nesting of loops must always be well-defined.

```
1   if (a == x)
2     if (a < b)
3       c = a;
4     else
5       c = b;
6   if (c != x)
7     ...
```

Figure 7: An example of C code which is improved by ϕ-elimination

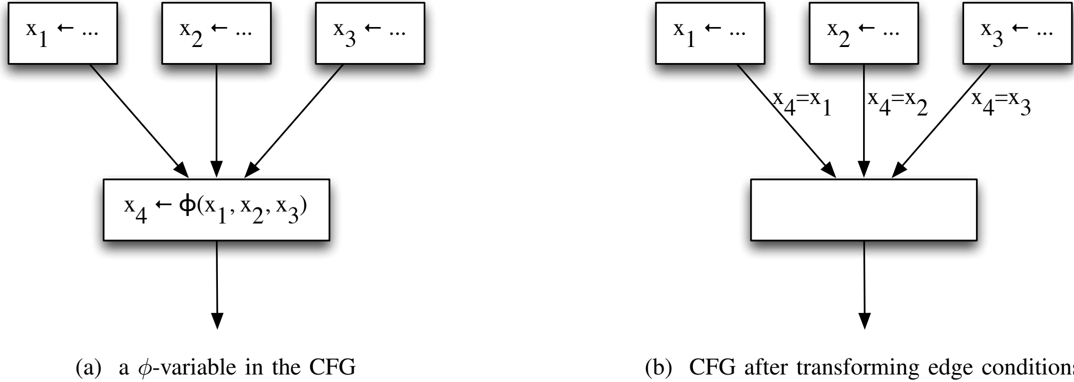
It would also be possible for Trickle to detect infeasible paths which involve the first iteration of a loop and any related conditions prior to entry of the loop body, and similarly paths involving the final iteration of a loop and any related conditions after the loop exit. We have not implemented this in our analysis, but it could detect infeasible paths which reduce the computed WCET estimate in some cases.

Model checkers are better suited to exploring iterations of a loop, however any resulting infeasible paths may not be expressible as ILP constraints. One simple method to simultaneously overcome the limitation of loops in SMT solvers, and the problem of constructing corresponding ILP constraints, is to (partly or fully) unroll loop iterations. We have not done so in our analysis, but this is also an area for future work. Loop unrolling is impractical though if the number of loop iterations is large or unknown. In some cases loop unrolling can also cause a state explosion if the loops contain conditionals, as the number of possible paths is then exponential in the number of iterations.

ϕ-elimination: In SSA representation, ϕ-functions (and the ϕ-variables to which they are assigned) are used to represent a value that depends on the path taken through the program. They are located at nodes in the control flow graph with multiple incoming edges. For example, Figure 6a shows part of a control flow graph where a ϕ-function is used. In the general case, the value of x_4 here depends on which incoming edge was taken. However, as our algorithm evaluates the feasibility of a specific path, we can transform the ϕ-function into edge conditions on the incoming edges, as shown in Figure 6b. This allows the SMT solver to track the variable through the path.

The process of ϕ-elimination actually gives rise to some of the "triple-diamond" scenarios described in Section II (although such scenarios can exist without ϕ-elimination). To see how this arises in practice, consider the code in Figure 7. Any loop-free path that executes both lines 3 and 7 is infeasible. We have not shown the full control flow graph for this example, but there are three edge conditions in the infeasible set: $\{a = x, c = a, c \neq x\}$.

Curbing complexity: Due to the inherently exponential nature of SMT solving, and the exponential complexity of the CAMUS algorithm in the presence of multiple MUSes,



(a) a ϕ-variable in the CFG

(b) CFG after transforming edge conditions

Figure 6: An example of ϕ-elimination

we must choose our candidate edges to test with CAMUS carefully. For programs with longer paths (e.g. 200 edges), there may easily be 10 or more MUSes on a given worst-case path.

To reduce the complexity of finding these MUSes, we observed that many conflicting edges are generally in close proximity. Given this, we use a small sliding window over the edges of the candidate path, and use CAMUS to find conflicts over smaller segments. The sliding window begins with a fixed initial size s_0, and moves forward incrementally by d for each query to CAMUS, partially overlapping with the previous query. This detects many of the common conflicts quickly. The variables s_0 and d can be tuned based on the efficiency of the SMT solver.

If no conflicts are found after the first pass with the sliding window, we repeat the process, increasing the size of the sliding window until it either contains an MUS or covers the entire path (at which point we know the SMT solver has deemed the path feasible). In extreme circumstances, the SMT solver may take too long as it performs a computationally expensive search. In this case, the impatient user can abort the SMT solver. This construction gives a suitable *anytime algorithm*, where the result returned is valid even if interrupted by the user prematurely.

As an example, consider a worst-case path that follows the sequence of edges:

$$e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8$$

with an initial window size of $s_0 = 4$, and increment of $d = 2$. We first run CAMUS over $\{e_1, e_2, e_3, e_4\}$, followed by $\{e_3, e_4, e_5, e_6\}$, followed by $\{e_5, e_6, e_7, e_8\}$. If no MUSes are found, we then expand to $\{e_1, e_2, \ldots, e_6\}$ and finally, $\{e_1, e_2, \ldots, e_8\}$.

Note that we may not find all MUSes along a specific path, but it is not necessary. For example, if a path contains one MUS with edges close together (close enough to fit within a small sliding window), as well as a second MUS where the conflicting edges are far apart, we will only find the first MUS. This eliminates the path in question (and any such paths containing the MUS) and guarantees forwards progress. We may later encounter the second MUS on a different worst-case path and can eliminate it then.

Using the sliding window technique, this approach could also be parallelized easily by processing different fragments of the sliding window on separate cores or machines.

Trickle also implements a further optimization, that removes redundant clauses before applying the CAMUS algorithm. Many paths often share the same edge condition amongst several edges. In such cases, only one instance of the edge condition needs to be passed to the SMT solver.

V. Evaluation

A. *seL4*

We demonstrate the applicability of Trickle by applying it to a worst-case interrupt response time analysis of the seL4 microkernel [5]. seL4 uses an event-driven non-preemptible kernel design, which ensures that no exceptions or interrupts occur within the kernel. This design was necessary to make formal verification feasible and also significantly simplifies static analyses of the kernel. However, it also increases the worst-case interrupt response time of the kernel. In order to bound interrupt response time and provide real-time responsiveness, *preemption points* are inserted into the kernel in long-running operations. A preemption point detects if an interrupt is pending, and if so, postpones the current operation to be resumed later, and handles the interrupt.

Our version of seL4 contains 125 non-preemptible regions of code. Each region begins and ends at program points where interrupts can be handled immediately—i.e. at a preemption point or kernel entry/exit. The worst-case interrupt response time of the kernel is determined by the maximum worst-case execution time of any of these 125 non-preemptible regions. In essence, we are running 125 separate WCET analyses, covering all code paths within the kernel.

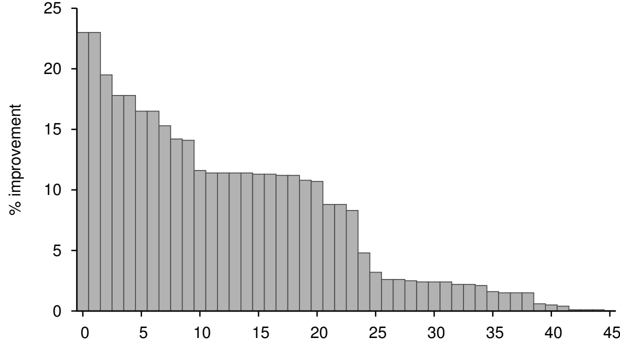


Figure 8: Percentage improvement of the 45 cases where our algorithm improved the worst-case execution time estimates by eliminating infeasible paths, compared with using no infeasible path information

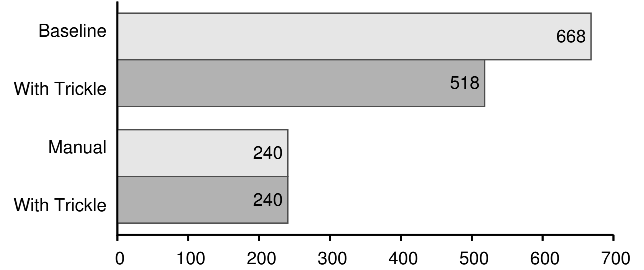


Figure 9: Reduction in WCET (in thousands of cycles) through applying automated infeasible path detection. The *baseline* figure uses no infeasible-path information, while *manual* is the best result achieved with manual annotations.

Trickle detects infeasible paths in 45 of the 125 regions. Where infeasible paths are found, Trickle arrives at a feasible path after one iteration in 31 cases, after two iterations in a further 14 cases, and up to 7 for the remaining 3 cases. In two cases, it reduces the WCET automatically by 23% (from 17 912 cycles to 13 800) using two iterations and 5 MUSes. The improvements in WCET for all 45 cases where Trickle detects infeasible paths are shown in Figure 8.

Trickle detects a total of 252 MUSes when applied to seL4. Note that these are only the MUSes required to identify specific worst-case execution paths as infeasible. Many more may exist in the program, but they do not affect our WCET computation. Of the MUSes found, 20 % are a single-element MUS (an infeasible basic block), 44 % are pair-wise conflicts, 30 % are 3-way conflicts, and 6 % are 4-way conflicts.

In our previous analyses of the seL4 microkernel [16], [17], the process of identifying infeasible path constraints was purely manual. Identifying a single infeasible path (or class of paths) by hand could take between 5 minutes and several hours. It involved two phases: (1) understanding what the execution path is doing and deciding if it is actually feasible; and if not (2) devising an infeasible path constraint expressible to the ILP solver which eliminates the path or class of paths. This process would need to be repeated until no infeasible paths could be observed. For a single non-preemptible region in seL4, this typically took several days of labour, as the path may initially be 100,000s of instructions long. The paths eliminated were manually identified as infeasible because of either local constraints, or global kernel invariants. Trickle only automates the former category, and as such the largest tangible benefit is the reduced human effort. This automation also made it practical to perform the WCET analysis over all 125 preemption points.

To compare the improvements over our manual analysis, we applied Trickle to the kernel binary used in our previous analysis [17]. The results are shown in Figure 9. Beginning with no infeasible path information (“baseline”), the reduction in WCET from Trickle is similar to what was obtained manually and previously validated by Sequoll [3], but with much less tedium. Trickle was unable to improve upon our best manual efforts (“manual”) to eliminate infeasible paths (which have not been machine-validated).

The difference between the results from Trickle and our best manual efforts is due to additional constraints which could not be deduced by sequoll. Over half of these extra constraints could be deduced if the memory aliasing analysis in sequoll was improved (currently, data written to heap memory are not tracked, so subsequent reads return unknown values). The remaining constraints rely upon global invariants maintained across the kernel (e.g. the type of certain objects found when dereferencing pointers). These latter constraints could not be deduced without reproducing a substantial portion of the seL4 proof. However, integrating such invariants directly from the proof is an interesting area for future research.

Pairwise-conflict comparison: We briefly explore the effectiveness of Trickle’s ability to find MUSes of size greater than 2 (e.g. the triple-diamond from our motivating example). We found that for the main kernel entry point, using only pair-wise edge conflicts gave a 4.7% improvement in computed WCET (compared with no infeasible path detection), whereas using n-way conflicts gave a 10.7% improvement. This difference indicates that eliminating infeasible paths with n-way conflicts can give notable improvements to WCET estimates.

B. Comparing CAMUS vs built-in unsat core

We can demonstrate the effectiveness of the CAMUS algorithm by comparing it with the single unsatisfiable core generated by our SMT solver (Yices 1.0.37). With either the CAMUS-generated MUSes, or the single “unsat” core provided by Yices, we are guaranteed to arrive at the same

Benchmark	# conflicts	# iter.	original WCET	new WCET	% diff
`cover`	3	1	20 781	18 283	-12.0%
`crc`	5	1	177 586	176 072	-0.9%
`statemate`	7	2	51 768	51 614	-0.2%
`ndes`	2	1	447 116	447 097	-0.0%

Table I: Infeasible paths detected in Mälardalen WCET benchmark suite

result, assuming that there are no bugs in the SMT solver.[1] What we can evaluate is how much faster we arrive at a feasible path.

Applying the CAMUS algorithm to seL4 required 67 refinements of the control flow graph. In comparison, using the unsat core generated by Yices required 152 refinement iterations. However, the number of refinement iterations alone does not necessarily imply a faster result, as it does not account for the extra run time of the CAMUS algorithm.

Our current implementation of CAMUS (and also the unsat core approach) does not immediately lend itself to a fair comparison. For technical reasons, a C program encoding the SMT problem is compiled for each SMT instance. A sizable proportion of the execution time is spent in the C compiler. An optimized implementation would not incur these overheads—e.g. by using direct API calls to construct the SMT problem from within Trickle.

To compare the execution times of CAMUS against the unsat core approach, we subtract out the time spent in compilation of the SMT instances, leaving only the run time of the SMT solver and Trickle itself. We note that in all of our experiments, we let the SMT solver run to completion.

The work required to compute the edge conditions is the same for both the CAMUS and unsat core approaches, and takes approximately one hour to compute across the entire seL4 binary using the Sequoll framework. Once all edge conditions are computed, using CAMUS takes 105 minutes to analyse all worst-case paths through seL4, whereas using a single unsat core requires 111 minutes. Note that these are the times after subtracting the compilation overheads—including the compilation time makes the unsat core method significantly slower because it uses many more SMT instances.

C. WCET benchmarks

We applied our algorithm to detect infeasible paths in programs from the Mälardalen WCET benchmark suite [6]. Like in our previous results from Sequoll [3], we had to omit benchmarks using floating-point arithmetic, irreducible loops and recursion, due to lack of tool support. Due to implementation shortcomings we were also unable to analyse several other benchmarks (in particular, our implementation does not currently handle 64-bit arithmetic instructions used in the compiled assembly of these benchmarks).

We detected infeasible paths in four benchmarks, shown in Table I. Depending on the benchmark, the number of infeasible paths detected and the improvement in WCET vary dramatically, and most have little impact. Although some of the Mälardalen benchmark programs have quite complex control flow graphs, most of them are comparatively small (compared to seL4). Infeasible paths are much more likely in larger programs, where they can cause significant overestimation.

Finally, we note that although many of the benchmarks are single-path programs (including `cover` and `crc`), they can still exhibit infeasible paths due to the abstractions used.

We see much larger improvements in seL4's WCET than in these benchmarks because of the increased complexity of the code being analysed in seL4. The size and structure of the seL4 code means that there are many infeasible paths which can be eliminated. For example, seL4 contains several functions that use switch statements over an object type [17]—for two such functions where one calls another, a quadratic number of infeasible paths are introduced, which are detected and eliminated by our algorithm.

VI. Related work

The issue of infeasible path detection in static analysis arises not only in the worst-case execution time domain, but also when generating test vectors [18], detecting programming errors [19], and in other general data flow analyses [20]. This has driven much research into finding infeasible paths automatically. However, the domain of worst-case execution time analysis, and specifically using IPET-based analyses, places many limitations on what infeasible path information can be utilized. Not all techniques for finding infeasible paths can be applied to IPET.

Engblom & Ermedhal demonstrate how various types of flow information can be converted to equations suitable for IPET-based analyses [14]. Like Trickle, they divide a program into "scopes," which are loop-free subsets of code, and express a number of different constraint types as ILP equations. These include the same construction we use for expressing conflicting edges. Their method also supports the ability to restrict constraints to specific loop iterations. They do not detect any constraints, but allow a user to provide these annotations more easily.

Suhendra et al. developed an algorithm to find infeasible paths for WCET analyses, based on detecting pairwise conflicts between assignments and conditional branches [21]. Like Trickle, they are also limited to finding such conflicts within loop-free CFGs, and thus treat each loop individually. As they only search for pairwise conflicts, their algorithm would not detect the case given in our motivating example.

Gustafsson et al. have shown that abstract interpretation can be combined with symbolic execution to find infeasible

[1] We encountered one bug in Yices which caused it to crash on certain inputs. The Yices developers swiftly fixed this bug in the 1.0.37 release.

paths [22]. Their approach is able to detect paths with more than two conflicting edges, and is similarly limited to loop-free segments of code. As their analysis is based upon abstract interpretation, a *join* can lose precision (but is necessary to curb complexity). They are limited by the reasoning ability of their symbolic execution engine (they present an interval-based solver), whereas we are limited by the reasoning ability of our SMT solver. In the absence of joins (no precision is lost), and if the reasoning ability of their symbolic execution engine was comparable to our SMT solver, then we would expect the infeasible paths found by their approach to be the same as Trickle. However a different complexity trade-off is made. Their method attempts to find all possible sets of infeasible paths in a single pass, using abstract joins to curb complexity, whereas Trickle takes a guided approach based on the current worse-case path in an iterative refinement. Given this, we expect that Trickle can perform better on loop-free code segments with an exponential number of flow facts.

Similarly, Ferdinand et al. perform a value analysis based on abstract interpretation to compute an interval of possible values for each register at each program point [23]. Their approach is also susceptible to loss of precision when paths join and, due to the use of intervals, has less expressive power than an SMT solver.

The model-checking-based approach taken by Cassez [24] to compute WCET inherently eliminates infeasible paths, as it is guaranteed to find a concrete worst-case (if the analysis terminates). However, the technique does not scale to programs the size of seL4.

Huuck et al. present a method to eliminate infeasible paths for reducing false positives in static analysis of source code [19]. They use a model checker to identify a path to an error condition, compute a weakest precondition from the path, and use an SMT solver to test the validity of the precondition. If the SMT solver shows that the precondition is unsatisfiable, they augment the model with an "observer" which encodes the unsatisfiability information and eliminates the path from subsequent model checking runs. Their approach works on loops, as the loop is detected by the model checker, and unrolled inside the weakest precondition expression given to the SMT solver. However, the infeasible path information they detect cannot be incorporated into an IPET-based WCET analysis.

Banerjee et al. demonstrate a method to incorporate information from infeasible path analysis into the micro-architectural model of the system [25]. They use a SAT solver to identify infeasible paths in parallel with the micro-architectural modelling phase to eliminate spurious machine states which arise due to the merging of paths in abstract interpretation, thereby reducing overestimation. This technique complements an existing infeasible path detection algorithm.

Our approach shares conceptual similarities with counterexample-guided abstraction refinement (CEGAR) algorithms for model checking [26], which were later extended to software verification [27]. CEGAR-style algorithms use counterexamples to refine the abstraction of a system in order to arrive at a model with the required precision to solve the problem. Černý et al. have applied CEGAR-like algorithms to general quantitative properties such as WCET [28]. They demonstrate automated abstraction-refinement schemes of cache behaviour, using standard CEGAR techniques for eliminating infeasible paths. Their approach however does not easily integrate into IPET-based analyses.

VII. Conclusions and Future Work

Eliminating infeasible paths is necessary to obtain good worst-case execution time estimates of large programs. In this paper, we introduced the Trickle algorithm to automatically detect infeasible paths within a control flow graph using an SMT solver. The computational complexity of the SMT solver is mitigated by iteratively analysing the worst-case path found so far. We integrate the CAMUS algorithm for finding all subsets of mutually unsatisfiable constraints to speed up the elimination of infeasible paths by reducing the number of iterations required, and we have shown how to integrate CAMUS effectively in the context of infeasible path detection.

Trickle can detect arbitrary sets of conflicting edges on a control flow graph. The information about infeasible paths can be easily integrated with an IPET-based WCET analysis. Being based on an SMT solver, it can detect more complex conflicts relations than other approaches.

We have evaluated our detection method by applying it to a worst-case execution time analysis of seL4, where we demonstrate that it can reduce the WCET estimate by 23%. By using the CAMUS algorithm, we can achieve these results with fewer iterations than using a single unsatisfiable core obtained from an SMT solver. We also significantly reduce the amount of labour, and potential for error, that arises from performing infeasible path elimination manually.

Future work includes investigating better handling of infeasible paths across loop iterations, either through unrolling or deducing loop invariants. The addition of a value analysis phase, using techniques such as abstract interpretation, may also improve the precision of the analysis in cases where values are propagated between loop nests. In addition, better memory aliasing analysis in the Sequoll framework would enable Trickle to detect more infeasible paths.

Trickle can be downloaded from http://www.ssrg.nicta.com.au/software/TS/wcet-tools/

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

REFERENCES

[1] Y.-T. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *16th RTSS*, 1995, pp. 298–307. 1, 3

[2] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comp. Progr.*, vol. 69, no. 1–3, pp. 35–45, Dec 2007. 1

[3] B. Blackham and G. Heiser, "Sequoll: a framework for model checking binaries," in *RTAS*, Eduardo Tovar, Ed., Philadelphia, USA, Apr 2013, Conference Paper, pp. 97–106. 1, 7, 8

[4] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *JAR*, vol. 40, pp. 1–33, 2008. 1, 2, 3

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *SOSP*. Big Sky, MT, USA: ACM, Oct 2009, pp. 207–220. 1, 6

[6] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *10th WS Worst-Case Execution-Time Analysis*. Brussels, Belgium: OCG, Jul 2010, pp. 137–147. 1, 8

[7] 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. Emb. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008. 2

[8] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, vol. 69, no. 1–3, pp. 56–67, Dec 2007. 3, 4

[9] A. Nadel, "Boosting minimal unsatisfiable core extraction," in *2010 FMCAD*. Lugano, Switzerland: FMCAD Inc, 2010, pp. 221–229. 3

[10] J. Bailey and P. J. Stuckey, "Discovery of minimal unsatisable subsets of constraints using hitting set dualization," in *7th PADL*. Long Beach, CA: Springer-Verlag, 2005, pp. 174–186. 3

[11] B. Dutertre and L. de Moura, "The Yices SMT solver," http://yices.csl.sri.com/tool-paper.pdf, SRI International, Aug 2006, visited 17 March 2013. 4

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *Trans. Progr. Lang. & Syst.*, vol. 13, pp. 451–490, October 1991. 4

[13] K. R. M. Leino, "Automating induction with an SMT solver," in *13th Int. Conf. Verification, Model Checking & Abstract Interpretation*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 315–331. 5

[14] J. Engblom and A. Ermedahl, "Modeling complex flows for worst-case execution time analysis," in *21st RTSS*, 2000. 5, 8

[15] R. E. Tarjan, "Testing flow graph reducibility," *J. Comp. & Syst. Sci.*, vol. 9, no. 3, pp. 355–365, 1974. 5

[16] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *32nd RTSS*, Vienna, Austria, Nov 2011, pp. 339–348. 7

[17] B. Blackham, Y. Shi, and G. Heiser, "Improving interrupt response time in a verifiable protected microkernel," in *7th EuroSys*, Bern, Switzerland, Apr 2012, pp. 323–336. 7, 8

[18] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," in *Information and Software Technology*, vol. 50, no. 7–8, 2008, pp. 641–655. 8

[19] R. Huuck, A. Fehnker, M. Junker, and A. Knapp, "SMT-based false positive elimination in static program analysis," in *ICFEM*, Kyoto, Japan, Nov 2012, Conference Paper, pp. 316–331. 8, 9

[20] R. Bodìk, R. Gupta, and M.-L. Soffa, "Refining data flow information using infeasible paths," in *Software Engineering ESEC/FSE'97*, ser. LNCS, vol. 1301. Springer Berlin Heidelberg, 1997, pp. 361–377. 8

[21] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Efficient detection and exploitation of infeasible paths for software timing analysis," in *43rd DAC*. New York, NY, USA: ACM, 2006, pp. 358–363. 8

[22] J. Gustafsson, A. Ermedahl, and B. Lisper, "Algorithms for infeasible path calculation," in *6th WS Worst-Case Execution-Time Analysis*, 2006. 9

[23] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *1st EMSOFT*. London, UK, UK: Springer-Verlag, 2001, pp. 469–485. 9

[24] F. Cassez, "Timed games for computing WCET for pipelined processors with caches," in *11th Int. Conf. Applic. Concurrency to Syst. Design*. Comp. Soc., Jun 2011. 9

[25] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "Precise micro-architectural modeling for WCET analysis via AI+SAT," in *19th RTAS*, Philadelphia, USA, Apr 2013, Conference Paper. 9

[26] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep 2003. 9

[27] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST: Applications to software engineering," *Int. J. Softw. Tools for Technology Transfer*, vol. 9, no. 5, pp. 505–525, 2007. 9

[28] P. Černý, T. A. Henzinger, and A. Radhakrishna, "Quantitative abstraction refinement," in *40th POPL*, 2013. 9