Using a Model Checker to Determine Worst-case Execution Time

Sungjun Kim
Department of Computer Science
Columbia University
New York, NY 10027, USA
skim@cs.columbia.edu

Hiren D. Patel
Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720, USA
hiren@eecs.berkeley.edu

Stephen A. Edwards
Department of Computer Science
Columbia University
New York, NY 10027, USA
sedwards@cs.columbia.edu

***Abstract*—Hard real-time systems use worst-case execution time (WCET) estimates to ensure that timing requirements are met. The typical approach for obtaining WCET estimates is to employ static program analysis methods. While these approaches provide WCET bounds, they struggle to analyze programs with loops whose iteration counts depend on input data. Such programs mandate user-guided annotations. We propose a hybrid approach by augmenting static program analysis with model-checking to analyze such programs and derive the loop bounds automatically. In addition, we use model-checking to guarantee repeatable timing behaviors from segments of program code. Our target platform is a precision timed architecture: a SPARC-based architecture promising predictable and repeatable timing behaviors. We use CBMC and illustrate our approach on Euclidean's greatest common divisor algorithm (for WCET analysis) and a VGA controller (for repeatable timing validation).**

***Keywords*-Real-time; Worst-Case Execution Time; Precision-Timed; Model-Checking**

I. Introduction

Software for hard real-time systems must satisfy strict timing constraints. This is typically done by determining the worst-case execution time (WCET) of programs through static program analysis methods. Such methods, however, have limitations. Due primarily to undecidability issues, only restricted forms of programs are statically analyzable. A large body of WCET research focuses on reducing the restrictions on programs, but we find that some striking restrictions still remains. One striking restriction: programs with dynamically changing loops bounds are not amenable to static program analysis because bounds for such programs cannot be automatically derived. Most WCET methods, therefore, require users to annotate the expected bounds [20].

In this paper, we address the issue of determining loop bounds automatically for programs with dynamically changing loop indexes. We present an automatic WCET analysis for the PRET architecture that uses a combination of static analysis for low-level analysis to extract execution time of basic blocks and model-checking to determine the dynamically changing loop bounds. We illustrate this method via a simple example with a complex data-dependent loop bound: Euclidean's algorithm for computing the greatest common divisor (GCD). The PRET architecture is unique in that it allows programs to specify timing requirements explicitly through instruction-set extensions. For example, through the use of timing instructions, it is possible to specify a periodic rate of execution of a program segments. To guarantee the periodicity, we again use CBMC [11] to validate that program segments enclosed in timing instructions indeed enforce repeatable timing behaviors. We illustrate this with a VGA driver example.

A. An Example: GCD

Figure 1 illustrates our techique. Figure 1(a) is a C implementation of Euclid's greatest common divisor (GCD) algorithm, which repeatedly subtracts the larger of two numbers from the other. While this is an uncommon algorithm to find in a real-time setting, it illustrates the challenges of calculating worst-case execution time when loops have complex behavior. Here, the values of the loop control variable b do not follow a simple pattern because they interact with the variable a.

How many times the loop in GCD iterates is a complicated function of a and b. In certain cases, the worst-case behavior is easy to see by inspection: if $1 \leq a, b \leq 100$, the worst case (100 iterations) occurs when $a = 1$ and $b = 100$. However, when $10 \leq b \leq 28$ and $70 \leq a \leq 94$, the maximum number of iterations is less obvious:[1] 31, which occurs when $a = 85$ and $b = 28$.

Figure 1 shows how we transform the C source program into a form that the Carnegie Mellon Bounded Model Checker (CBMC) can analyze, which we use to calculate the WCET.

Our basic strategy is to add a variable to the program whose current value represents the number of clock cycles consumed by the program to reach that point and ask the model checker what values this variable may take. For example, if we know the variable may not exceed a certain value, that value is a bound on the WCET.

In the beginning of the WCET analysis, the input data range is written in the source code. In the GCD example, the input parameters, a and b, are passed from arrays, of which values ranges from 1 to 100.

After that, we divide the program into basic blocks and add a statement at the beginning of each block that increases the time variable according to the number of cycles taken by

[1] We found it by simply running the algorithm on every pair of numbers in the range.

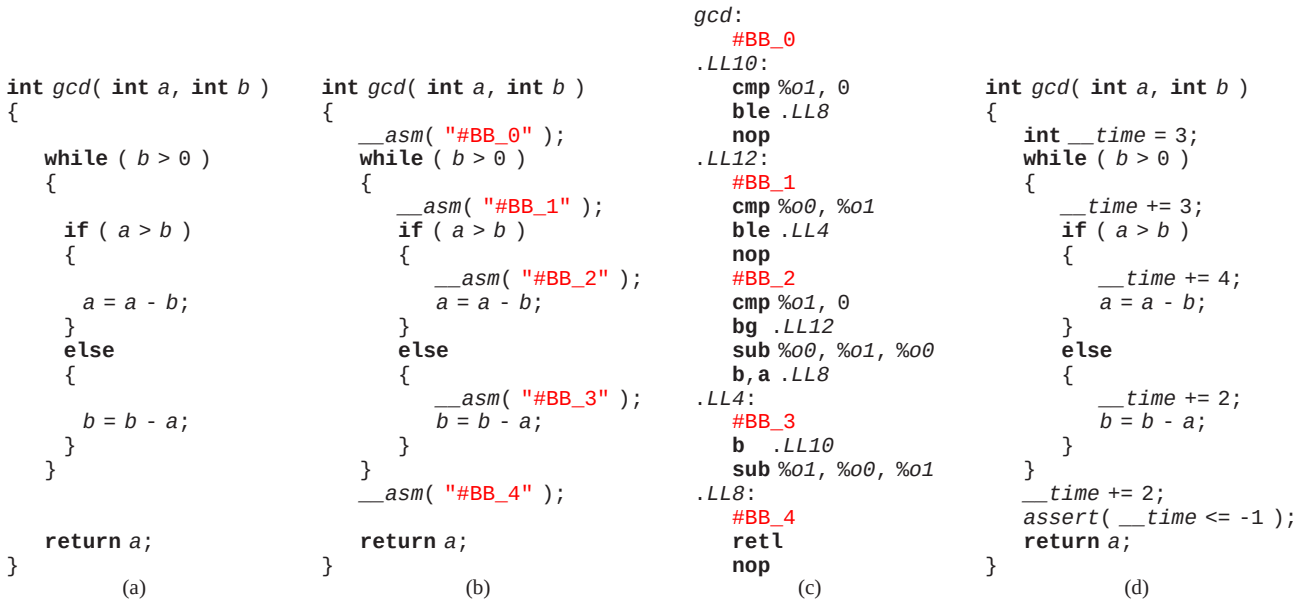

Fig. 1. (a) Euclid's greatest common divisor algorithm. Our technique (b) inserts comments at basic block boundaries, (c) compiles it to assembly (with optimization), and (d) backannotates the execution time of each basic block to produce a C program suitable for the model checker.

the block. This is possible because we assume a predictable architecture whose statements do not interact temporally [5], [15]; the technique would be difficult to adapt for architectures with much more interaction.

Ultimately, it is the compiler that determines the instructions in each basic block, but the compiler does not modify the block structure itself. Taking advantage of this, we start by inserting an assembly comment at the beginning of each basic block (Figure 1(b)). The compiler passes these through unmodified, allowing us to locate the basic blocks in the assembly it generates (Figure 1(c)).

After counting instructions in each basic block in the assembly source, we back-annotate the original C source with instructions that update the time variable (Figure 1(d)) and hand the result to CBMC for analysis.

We call CBMC repeately to compute the WCET. We treat the model checker as a black-box that takes the program, an estimate of WCET, and the unrolling depth as inputs and may return one of three results. It may tell us that the unrolling depth is insufficient (i.e., the function is still running after the requested number of iterations), in which case we double the unrolling depth and try again.

Another possibility: the unrolling depth may be sufficient but the property may be false. For example, the WCET of this example is clearly not -1. In this case, CBMC provides a counterexample (i.e., a reachable state in which the execution time is greater than -1), we use it as the new estimate of WCET (i.e., change the *assert* in Figure 1(d)), and run the model checker again.

Finally, the unrolling depth may be sufficient and the property is true (i.e., the *assert* on the timing is always true). In this case, we terminate and report the WCET. We iterate this process until we run out of memory or patience.

B. The PRET Philosophy

It is no surprise that most abstractions in computing hide timing properties of software. This has the immediate advantage that computer architects can use clever techniques to improve the average-case performance through architectural optimizations such as speculative execution, deep pipelining, and complex memory hierarchies. This, however, comes at the expense of predictable and repeatable timing behaviors making the task of calculating accurate execution bounds of a sequence of instructions extremely difficult [6]. While this is acceptable for general purpose computing, the absence of time is severely detrimental to computing systems where timeliness is just as critical as correct functionality. Clearly, hard real-time embedded computing is such an example. As real-time embedded computing continues to borrow computer architectures, tools, and techniques from general purpose computing, we find that this results in computing systems that are unpredictable, non-repeatable, and brittle.

Consequently, the philosophy behind PRET [5] is to make temporal characteristics of real-time embedded computing just as predictable, and repeatable as function. In particular, PRET's objective is to re-think many of the architectural features, and to judiciously select the ones that deliver performance enhancements, but without sacrificing predictable and repeatable timing behaviors. Our initial prototype [15] of the PRET architecture employs software-managed scratchpad memories [1], thread-interleaved pipelines without bypassing [12], time-triggered arbitration access to off-chip memory, explicit control over timing behaviors through ISA extensions [10], and high-level language extensions for specifying timing requirements [19].

By design, the PRET architecture makes predicting exe-

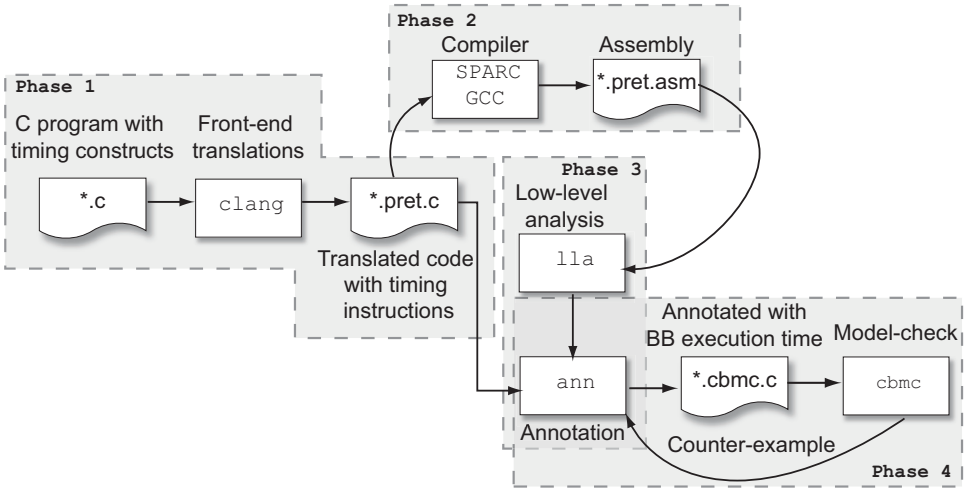

Fig. 2. Design Flow using Both Static Analysis and Model-checking

cution times of instructions on the processor straightforward. This is because every instruction is completely predictable allowing us to construct a simple yet a precise hardware architecture model to estimate the execution times of instructions. To control timing behaviors in software the PRET ISA provides *deadline instructions* [5], [15]. A deadline instruction sets cycle counters to a given value and continues decrementing while executing the instruction stream. If another instance of the deadline instruction is reached in the instruction stream before the counter is zero, the program's execution is stalled until it is zero before reloading a new value and continuing the processing of the instruction stream. This guarantees that a certain segment of program code repeatably takes the same amount of execution time.

II. Related Work

At present, integer linear programming (ILP) [14], and implicit path enumeration technique (IPET) [13] combined with value analysis [3] and pipeline, and cache analysis [20] are popular techniques used in estimating WCET [17], [8]. However, these methods raise several issues. For example, IPET analysis requires the users to annotate the loop bounds, and back-edges of a control flow graph to formulate an ILP. While value analysis combined with control flow analysis (CFA) [7] is often successful in providing loop bounds, it only supports a small subset of programs. This leaves a large number of data-dependent programs uncovered with this method. One such example program is the GCD algorithm, for which we devise a solution that uses model-checking.

The the general concept of WCET analysis through model checking was proposed by Metzner in [18]. However, that proposed method requires manually annotating loop bounds, which we eliminate in our work by unrolling the loops using bounded model-checking combined with an iterative deepening search. Methods that use symbolic representation [16] need users to determine the loop bounds manually. For large programs, this can be an extremely difficult and error-prone task. In contrast to such symbolic methods, our approach does not require the user to provide the loop bounds at all, but instead, we simply take the original C program, and automatically translate it into a program with annotation rules. State of the art commercial tools such as AbsInt [20] provide almost fully automated usability with code annotation, value analysis and IPET analysis. However, aside from user-specified annotations, they do not provide an automated method for data-dependent loop bounds.

III. Our Technique

Our approach has four phases to it as shown in Figure 2. The first phase converts a C program with timing constructs [19] to a standard C program that is compilable with GCC. During this phase, deadline instructions are automatically inserted into the source code in place of the timing constructs. Phase two simply describes compiling the translated source code using the SPARC GCC toolchain. The resulting assembly code is used as input in phase 3 for the low-level analysis (`lla`) that determines execution time of each basic block. The translated source code from phase 1 is annotated with the execution times of basic blocks (`ann`). In addition, we add an assertion for the WCET of the program. The resulting source code with annotations (analyzable code) is sent to CBMC for assertion verification. Given that we get a counter-example, we repeat the process of updating the assertion with a new value for the WCET until we have exhaustively searched all feasible possible paths, and computed the WCET.

A. Source Code Features

Input to our WCET analysis are C programs compilable with the SPARC GCC toolchain. In addition, Patel et al [19] have proposed timing constructs as language extensions to specify temporal requirements in software. An example of a timing construct, `DEADSEQ()`, is shown in Figure 3, which specifies that the program code within the `DEADSEQ()` scope should always take *N* units of time to execute. These timing

constructs are realized into deadline instructions specific for the PRET architecture. We use an open-source front-end `clang` to translate the C programs with timing constructs into its equivalent representation with the appropriate deadline instructions. Currently, we have preliminary support for the automated translation, and our ongoing efforts are in making this more robust.

```
DEADSEQ( N )
{
   while( <expr> )
   {
      <stmt>
   }
}
```
(a)

```
DEADPLLSEQ( N )
{
   while( <expr> )
   {
      <stmt>
   }
}
```
(b)

Fig. 3. Generating repeatable timing

There are two kinds of deadline timing constructs, as illustrated in Figure 3. Figure 3(a) is a deadline construct with thread clock synchronization. (cf. In PRET, 1 thread cycle is same as 6 CPU cycles [15].) The semantic is that the program code enclosed within the curly braces takes at least N thread cycles to execute. If the `while` loop takes shorter than N thread cycles, PRET stalls the execution until N thread cycles elapse. On the other hand, if it takes longer, the program goes to the next instruction without stalling. Likewise, Figure 3(b) is a deadline construct with phase-locked loop (PLL) clock synchronization. Its semantic is exactly the same except that the given argument N, is synchronized by a PLL clock.

Before starting the timing analysis, we define the ranges of input data. In the GCD algorithm, we use arrays as the input parameters, thus ranging the values of them is straightforward: with array size of 100, we set each value of an array element from 1 to 100.

B. Division into Basic Blocks

The initial step of our timing analysis is dividing the source code into basic blocks with assembly labels. First, we identify locations in the program code that represent individual basic blocks, and insert an assembly label, `__asm( "#BB_nr" );`. (cf. `nr` is a number that increases whenever a label is added.) The rules of inserting assembly labels are given in Figure 4. We describe the result for the GCD algorithm in Figure 1(b).

Next, we compile the labeled C code into assembly code, and extract an execution time for every basic block. Since instructions between the inserted assembly labels comprise a basic block, the running time of these instructions are the execution time of the basic block. In addition, all the instructions of the PRET processor are predictable, thus we use a simple table-driven assignment of the execution times to the basic blocks.

We illustrate this in Figure 1(c). The first basic block is from `#BB_0` to `#BB_1`; since it has only 3 non main-memory accessing instructions, the execution time of it is 3×1 thread clocks. (cf. in PRET, non main-memory accessing instructions take 1 thread clocks while main-memory accessing instructions take 15 thread clocks [15].)

```
if( expr ) stmt   →   if( expr )
                      {
                         __asm( "#BB_nr" );
                         stmt
                      }
```
(a)

```
if( expr ) stmt
else stmt         →   if( expr )
                      {
                         __asm( "#BB_nr" );
                         stmt
                      }
                      else
                      {
                         __asm( "#BB_nr" );
                         stmt
                      }
```
(b)

```
case const_expr:
stmt              →   case const_expr:
                      {
                         __asm( "#BB_nr" );
                         stmt
                      }
```
(c)

```
while( expr ) stmt   →   while( expr )
                         {
                            __asm( "#BB_nr" );
                            stmt
                         }
                         __asm( "#BB_nr" );
```
(d)

```
for(expr;expr;expr)
stmt              →   for(expr;expr;expr)
                      {
                         __asm( "#BB_nr" );
                         stmt
                      }
                      __asm( "#BB_nr" );
```
(e)

Fig. 4. Basic block dividing rules

C. Adding Execution Time Annotations

After dividing the source code into basic blocks with the assembly labels, we build our timing analyzable code with annotations. There are two kinds of annotations: inserting execution times into its basic blocks and adding assertion statements for the timing analysis. We annotate execution times of basic blocks as computed in section III-B and use `assert` statements as instructed in the CBMC manual. CBMC recognizes the `assert` statements and verifies the property inside it [11].

At first, we insert the execution times of all the basic blocks to the original code. The positions where assembly labels are inserted are also the places to add the execution times. Comparing Figure 1(b) and Figure 1(d), we can clearly notice this.

When we encounter timing constructs, we annotate the code as shown in Figure 5. Note the three comments, (1), (2) and (3). The annotations in these comments separate two different deadline scopes: `DEADSEQ( N1 )` scope and

```
DEADSEQ( N1 )
{

   <stmt>

   DEADSEQ( N2 )
   {

      <stmt>

   }

}
```
(a)

```
DEADSEQ( N1 )
{
   int __time = 0;     // (1)
   __time += T1;
   <stmt>
   __time += N2;       // (2)
   DEADSEQ( N2 )
   {
      int __time = 0; // (3)
      __time += T2;
      <stmt>
      assert( __time < N2 );
   }
   __time += T3;
   assert( __time < N1 );
}
```
(b)

Fig. 5. (a) Nested deadline constructs and (b) its annotated code

`DEADSEQ( N2 )` scope. The outer `DEADSEQ` scope consider the execution time of the inner `DEADSEQ` scope as $N2$ as marked in (2); however, since the inner should ignore the execution time declared in (1), the annotation in (3) is inserted.

The assertion statements are added at their proper positions: at the end of the target function of the WCET analysis (Figure 6(a)), or at the end of the deadline timing construct (Figure 6(b) or Figure 6(c)). Also, as illustrated, for WCET analysis the asserted property compares `__time` with -1 (Figure 6(a)). However, for validating whether timing requirements are satisfied for the timing constructs, we use the deadline argument in the assertion (Figure 6(b) or Figure 6(c)).

Validating the PLL timing constructs, we have to consider that the given deadline argument is synchronized with the PLL clock, which can be different from the thread clock (or the CPU clock). In this case, the assertion property, the deadline instruction's argument, is simply converted by multiplying the ratio of thread and PLL clocks. Suppose that the thread clock frequency is X and the PLL frequency is Y, as described in Figure 6(c). Since N is synchronized by the PLL clock, the assertion statement for the timing validation should be `assert( __time <= N * X / Y );`.

D. Running the Model Checker

After annotating the source code, we use the model checker, CBMC, to verify the property: the `assert` statements. For validation, we only have to model-check once, and if the property is verified, then the timing requirements are satisfied. On the other hand, for WCET analysis, we update the asserted property with the WCET revealed through the counter-example until the property is verified. The final annotated property is the WCET.

For model-checking, we use Carnegie-Mellon Bounded Model-Checker (CBMC) [11]. It effectively finds loop bounds and WCETs, and validates the deadline timing constructs. Since CBMC is a verification tool for ANSI-C/C++ programs, we can simply use the original PRET C source code to perform the timing analysis. Furthermore, we do not need to annotate loop bounds for our timing analysis. This is because the bounded model-checker [2], [4] verifies whether the loop bound is reached. Thus, the loop bound is determined via an iterative deepening search. Initially, we provide an arbitrary number of loop unwinds and run CBMC. If CBMC verifies that the loop terminates, then we have determined the loop bound. Otherwise, we increase the number of loop unwininding and run it again, and continue repeating these procedures until the loop bound is found.

To explain WCET analysis in detail, let's see Figure 6(a). Initially, we do not know the WCET nor the loop bound. Therefore, we start with finding the WCET with a small number of loop unwinds. Provided verification fails due to insufficient loop unwinding, we increase this number and re-run CBMC (finding the loop bound). If CBMC provides a counter-example, we change the assertion statement while guessing the WCET as the counter-example, and re-run CBMC. We repeating these steps until CBMC yields successful verification whereby the candidate WCET is the WCET. When we find the WCET, loop bound is also found because the model checker has exhaustively searched all the feasible paths.

We illustrate this method to compute the WCET of the GCD algorithm. We first start the candidate WCET as -1 with 10 loop unwinds. Whenever verification fails due to insufficient loop unwind, we double the number of unwinds. For two arguments ranging from 1 to 100, the result WCET is 703 thread cycles.

Unlike the WCET analysis, which requires multiple invocations of CBMC, validating repeatable timing behaviors needs performing the verification only once because the assertion explicitly checks for the timing requirement. A sample code is shown in Figure 6(b). Notice that we only need to unwind the loop at most N times. This is under the conservative assumption that the loop takes one thread cycle to execute. So, if the enclosed program code does not meet its timing requirement of N thread cycles with N number of unwindings, then the timing requirement is violated. This goes to show that in order to validate that program segments have repeatable timing behavior, we do not need to determine the loop bounds. Instead, we can unwind N times to validate our timing requirement. For these reasons, we don't have to precondition the loop termination for the timing validation.

Not only that, but the timing validation method is also particularly fast when the target code has nested deadline constructs and when the validation fails in an outer scope. Let's see Figure 5. Note that `__time` variable of the inner `DEADSEQ` is invisible to the outer `DEADSEQ` scope and the outer `__time` ignores the inner. Thus, timing validations of two `DEADSEQ`s work differently. Since timing properties of the inner `DEADSEQ` is ignored when `assert` in the outer `DEADSEQ` is verified, the execution time of the inner DEADSEQ is simply regarded as `deadline_inner`. The timing validation of the inner `DEADSEQ` is done only if that of the outer is passed; otherwise, the verification shortly finishes with failure. Thus, the repeatable timing validation of nested deadlines can be faster in a validation failure case.

```
int __time = 0;
void foo( )
{
   __time += T1;
   while( <expr> )
   {
      __time += T2;
      <stmt>
   }
   __time += T3;
   assert( __time < -1 );
}
```
(a)

```
DEADSEQ( N )
{
   int __time = 0;
   __time += T1;
   while( <expr> )
   {
      __time += T2;
      <stmt>
   }
   __time += T3;
   assert( __time < N );
}
```
(b)

```
DEADPLLSEQ( N )
{
   int __time = 0;
   __time += T1;
   while( <expr> )
   {
      __time += T2;
      <stmt>
   }
   __time += T3;
   assert( __time < N * X / Y );
} // X: thread clock frequency
  // Y: PLL frequency
```
(c)

Fig. 6. Time annotated code: (a) WCET analysis, (b) timing validation for a deadline construct and (c) timing validation for a PLL deadline construct

IV. Experimental Results

A. GCD Algorithm

Stage	*Unwind*	*Candidate WCET*	*Counterexample*
1	10	-1	10
2	10	10	20
3	10	20	50
4	10	50	59
5	10	59	Unwind Error
6	20	59	62
7	20	62	95
8	20	95	97
9	20	97	103
10	20	103	Unwind Error
11	40	103	125
12	40	125	237
13	40	237	283
14	40	283	Unwind Error
15	80	283	360
16	80	360	Unwind Error
17	160	360	492
18	160	492	505
19	160	505	507
20	160	507	540
21	160	540	610
22	160	610	682
23	160	682	701
24	160	701	703
25	160	703 (WCET)	Verification Success

TABLE I
Iterative deepening WCET analysis of the GCD algorithm

The WCET analysis result of the GCD algorithm is shown in Table I. The test starts with 10 unwinds, finding counter-examples. We double the number of unwinds whenever we encounter an unwind error, which means verification cannot complete with the given number of unwinds. We also substitute the candidate WCETs as the counter-example and the test continues until the verification succeeds. The WCET is 703 thread cycles for our GCD example, in which the input variables are ranging from 1 to 100.

B. VGA Driver

We use the VGA driver example presented in [15] to confirm our approach for validating timing requirements for the PRET architecture. The VGA driver sends four colors of pixel data to the VGA controller. The VGA timing requirements for 640×480 resolution with 60 Hz refresh rate must be met. There are blank timing and active timing requirements. The driver should send image data one by one according to the VGA clock, 25.175MHz, in active timings. To produce the VGA clock, we use the PLL clock, and to send image data, we load the data to a 32-bit hardware shift register.

The implemented VGA driver algorithm is a nesting of simple loops. The driver iterates every vertical timing, a frame drawing timing; a horizontal timing, a line drawing timing, iterates inside the frame timing while sending image data to the VGA controller. Each iteration must follow the VGA timing requirement that is synchronized by the PLL clock. Repeatable timing constructs are inserted for the iterations of vertical and horizontal timings.

Since the current PRET processor prototype is in the form of a simulator, we need to compute the CPU speed that allows correct operation of the VGA driver. In our VGA driver example, we calculate the minimum CPU speed manually, and then validate the driver code to determine whether any timing requirements are violated. As a result, the calculated minimum CPU speed is 188MHz. We use this frequency when we perform the validation, and verify that our timing requirements are satisfied. We also investigated reducing the frequency. This resulted in counter-examples, as we had expected.

V. Conclusion

This work presents a hybrid approach to WCET analysis. We leverage model-checking techniques to derive loop bounds and cover feasible program paths, and static program analysis to determine basic block execution times. By combining these two techniques, we are able to compute WCET of programs whose loop bounds are allowed to change dynamically. Note that we do require the user to provide the range of possible values that the loop arguments can take. Additionally, we validate that PRET's timing constructs meet their timing requirements using this approach. Our future work involves extending the WCET analysis for timing constructs that provide time-based exceptions [9], synchronization instructions between threads, and employing this WCET analysis for scratchpad memory allocation schemes [19].

REFERENCES

[1] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, California, January 1977.

[4] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *A Survey of Automated Techniques for Formal Software Verification*, 2008.

[5] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th Design Automation Conference*, pages 264–265, San Diego, California, June 2007.

[6] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, North Lake Tahoe, California, October 2001.

[7] Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A tool for automatic flow analysis of C-programs for WCET calculation. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 106–112, Guadalajara, Mexico, January 2003.

[8] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003.

[9] IEEE Real-Time and Embedded Technology and Applications Symposium. *Poster Abstract: Timing Instructions - ISA Extensions for Timing Guarantees*, April 2009.

[10] Nicholas Jun Hao Ip and Stephen A. Edwards. A processor extension for cycle-accurate real-time software. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 4096 of *Lecture Notes in Computer Science*, pages 449–458, Seoul, Korea, August 2006.

[11] Daniel Kroening and Edmund Clarke. Carnegie-Mellon Bounded Model-Checker (CBMC). `http://www.cprover.org/cbmc/`.

[12] Edward A. Lee and David G. Messerschmitt. Pipeline interleaved programmable DSP's: Architecture. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(9):1320–1333, September 1987.

[13] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, December 1997.

[14] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, Pisa, Italy, December 1995.

[15] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 137–146, Atlanta, Georgia, October 2008.

[16] G. Logothetis and Klaus Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 196–203, Munich, Germany, March 2003.

[17] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, page 12, Washington, DC, 1999. IEEE Computer Society.

[18] Alexander Metzner. Why model checking can improve WCET analysis. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347, Boston, Massachusetts, July 2004.

[19] Hiren D. Patel, Ben Lickly Ben, Bas Burgers, and Edward A. Lee. A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture. Technical Report UCB/EECS-2008-115, EECS Department, University of California, Berkeley, September 2008.

[20] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The determination of worst-case execution times: Overview of the methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, April 2008.