Parametric Timing Analysis for Complex Architectures*

Sebastian Altmeyer†
Department of Computer Science
Saarland University
altmeyer@cs.uni-sb.de

Christian Hümbert‡
AbsInt GmbH
Saarbrücken
huembert@absint.com

Björn Lisper
Department of Computer Science and Electronics
Mälardalen University
bjorn.lisper@mdh.se

Reinhard Wilhelm
Department of Computer Science
Saarland University
wilhelm@cs.uni-sb.de

Abstract

Hard real-time systems have stringent timing constraints expressed in units of time. To ensure that a task finishes within its time-frame, the designer of such a system must be able to derive upper bounds on the task's worst-case execution time (WCET). To compute such upper bounds, timing analyses are used. These analyses require that information such as bounds on the maximum numbers of loop iterations are known statically, i.e. during design time. Parametric timing analysis softens these requirements: it yields symbolic formulas instead of single numeric values representing the upper bound on the task's execution time.
In this paper, we present a new parametric timing analysis that is able to derive safe and precise results. Our method determines what the parameters of the program are, constructs parametric loop bounds, takes processor behaviour into account and attains a formula automatically. In the end, we present tests to show that the precision and runtime of our analysis are very close to those of numeric timing analysis.

1 Introduction

Determining upper bounds on the execution times of tasks is one of the most eminent challenges during the design of a hard real-time system; a task missing its deadline may cause the whole system to fail. Therefore, upper bounds of the worst-case execution times (WCET) must be known at design time.

Due to the significance of the topic, many research groups addressed it in the last years (see [12] for an overview). Several timing analysis approaches have been implemented and used in practice. Nevertheless, finding precise and safe timing guarantees is considered a complex and time consuming task. Furthermore, all data influencing the timing behaviour, such as the maximal number of loop iterations, must be known in advance, i.e. during the analysis. However, some systems need guarantees for timely reactions which are not absolute, but dependent on a numerical parameter. Examples are operating-system schedulers which schedule a fixed set of tasks and servers who process a number of requests. In such cases, there are only two possibilities: either provide bounds for the unknown variables or start a new analysis each time the task is used with different values. The first option endangers precision, the second may unacceptably increase the analysis time.

Parametric timing analysis is an extension of *numeric timing analysis*. Instead of computing a single numeric value for the WCET, a parametric analysis is able to derive symbolic formulas. The WCET for a task and a specific parameter assignment is then simply derived by evaluating the task's timing formula. Imagine again the scheduler of an operating system. This scheduler can be used within different embedded systems, each time with a different number of tasks. A parametric analysis is able to derive a WCET formula depending on this parameter. Therefore, a wider class of tasks may be analysed statically. Furthermore, a formula shows how the execution time depends on parameters; information which allows to adjust the parameters such that timing constraints are met.

In this paper, we propose a new method for parametric timing analysis. Our approach analyses executables to de-

*This work was supported by the European Community's Sixth Framework Programme as part of ARTIST2 Network of Excellence. See www.artist-embedded.org for more information.

†Partially supported by the German Research Council (DFG) as part of AVACS (SFB/TR 14) See www.avacs.org for more information.

‡Partially supported by the European Community's Sixth Framework Programme under grant agreement n° 33661.

rive safe upper bounds. A parameter is a variable whose value before program execution influences the program flow and so the program's execution time. Such a parameter is either stored in memory or in a register (and is thus visible to the user) or it is determined by the size of a dynamic data structure accessed within a loop or in a recursion. In the first case, the analysis automatically identifies the parameters. In the second case, we assume that the user has specified a parameter that bounds the number of iterations of the loop or recursion that traverses this data structure. So, the timing behaviour can be analysed with respect to this parameter. Note that from a technical view, the user annotation in the second case is a rather small improvement – therefore, we will mainly focus the first case here.

The contributions of our new method are the following:

- Our analysis operates on executables, thus analyses the actual instructions to be executed and can therefore compute precise and safe upper bounds.
- The method is able to perform the whole analysis automatically, starting from the identification of the parameters, determination of parametric loop bound expressions up to the derivation of symbolic timing formulas.
- The method takes the low-level behaviour of processors (e.g. caches, branch prediction) into account and thus computes valid upper bounds even for complex processors.
- We have implemented a prototype (targeting the PowerPC 565 and 755) to provide results of a parametric analysis and show the practical feasibility of our approach.

The remainder of the paper is structured as follows: we first give a short introduction to numeric timing analysis our approach is based on in Section 2. We describe our new parametric analysis in detail in Section 3, followed by practical tests and evaluation in Section 4. We compare our work to existing work in Section 5 and conclude in Section 6.

2 Timing Analysis - State of the Art

The timing of modern processors highly depends on caches, pipeline effects, branch prediction, etc. An analysis has to take these effects into account and has to resort to the level of the executable. We build our parametric timing analysis on top of the *aiT-Framework* [9] which analyses executables. The aiT-timing analysis as depicted in Figure 1 consists of a set of different tools that can be subdivided into three main parts:

- *CFG Reconstruction*
- *Static Analyses*
- *Path Analysis*

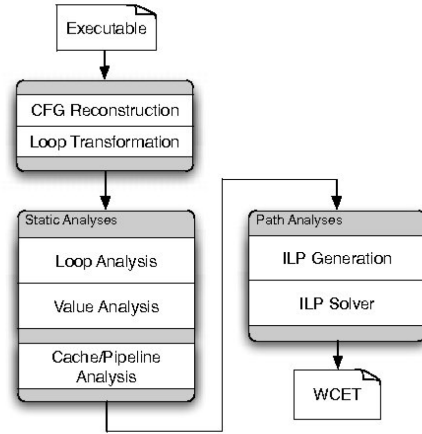

Figure 1. The aiT toolchain

The *CFG reconstruction* builds the control-flow graph (CFG), the internal representation, out of the binary executable [18]. This CFG consists of so-called *basic blocks*. A basic block is a list of instructions such that the basic block is always entered at the first and left at the last instruction. To make sophisticated interprocedural analysis techniques applicable, loop structures that the previously constructed CFG still contains, have to be transformed into tail-recursive routines. Additionally, user annotations, such as upper bounds on the number of loop iterations which the analysis can not automatically derive, are processed during this step.

The static-analysis part consists of three different analyses: *loop analysis*, *value analysis* and a combined *cache and pipeline analysis*. The *value analysis* determines the effective addresses of memory accesses and also supports the loop analysis to find upper bounds on the number of loop iterations [16, 5]. For this purpose, the analysis derives intervals for all variables within a program. Such an interval for a variable x consists of a lower bound $a \in \mathbb{Z} \cup \{-\infty\}$ and an upper bound $b \in \mathbb{Z} \cup \{\infty\}$ such that $a \leq x \leq b$ holds.

The *loop analysis* collects invariants for all potential loop counters. This means it computes for all variables changed within a loop, how much they change during one iteration. Then, it evaluates the loop exits, request start and end values for these potential loop counters from the value analysis and thus derives upper bounds on the number of loop iterations. If, for instance, a variable v is initialised by a constant c_{init}, increased by c_{inc} in each loop iteration and compared to constant c_{exit} (e.g., while $(v < c_{exit})\{\ldots\}$) at the loop exit, the loop is obviously executed at most $\lceil (c_{exit} - c_{init})/c_{inc} \rceil$.

The *cache and pipeline analysis* performs the so-called *low-level analysis*. It simulates the processor's behaviour in an abstract fashion to determine for each basic block an upper bound on its execution time [20, 10].

The *path analysis* combines the timing information for each basic block and the loop bounds and searches for the longest path within the executable. In this fashion, it computes an upper bound on a task's execution time. Searching the longest path is done using a technique called *implicit path enumeration (IPET)* [19]: the control flow graph and the loop bounds are transformed into *flow constraints*. The upper bounds for the execution times of the basic blocks as computed in the cache and pipeline analysis are used as weights. Figure 2 provides an example. The variables n_i, also called traversal counts, denote how often a specific edge is traversed. The first and the last basic block are left, resp. entered, exactly once ($n_1 = 1$; and $n_3 + n_6 = 1$;). For all other basic blocks, the sum of the traversal counts entering equals the sum leaving. The loop body (basic blocks 4, bounded by b_{loop}) is executed at most b_{loop} times as often as the loop is entered ($n_4 <= b_{loop} n_2$;). The constant c_j denotes the cost of the basic block j. The maximum sum over the costs of a basic block times traversal counts entering it determines the final WCET bound.

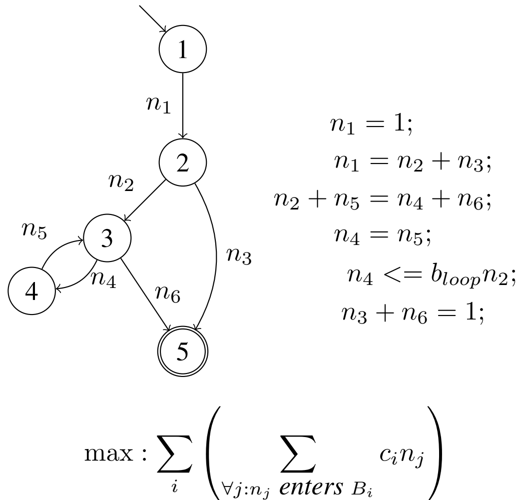

Figure 2. Control flow graph and the corresponding flow constraints

3 Parametric Timing Analysis

A parametric analysis on the executable level has to face several problems. One has to determine:

- the parameters of the program to be analysed,
- the influence of these parameters on the program's execution, i.e. the parametric loop bounds,[1]
- the symbolic timing formula using implicit path enumeration.

Since we are interested in minimising the amount of user annotations, we have to implement automatic analyses for the first two parts. For the last part, we have to find a symbolic optimisation method which is able to derive formulas instead of numeric values. However, these three parts can not be solved in isolation, since a solution for one of the three parts clearly influences the others.

Figure 3 denotes the structure of the parametric timing analysis. The new or changed parts with respect to the non-parametric analysis are marked bold. First, the parameter analysis determines the parameters of the program to be analysed and the variables that depend on these parameters. Second, the loop analysis computes loop bounds for the non-parametric loops and loop-bound expressions for the parametric loops. The cache and pipeline analysis remains unchanged. Later on, in the parametric path analysis, a *symbolic integer linear program* [8] is generated and solved.

Such a symbolic ILP finds the optimum of a linear function over the following set:

$$\{\vec{x} | \vec{x} \geq \vec{0}, A\vec{x} + B\vec{z} + \vec{c} \geq \vec{0}, \vec{x} \text{ integral}\}$$

where $\vec{z}$ is a vector of parameters. The result is a conditional expression in this vector $\vec{z}$. Consider again the example given in Figure 2. The vector $\vec{x}$ represents the variables n_1 to n_6, vector $\vec{z}$ represents the parameter b_{loop} and the linear function is given by $\max : \sum_i \left(\sum_{\forall j : n_j \textit{ enters } B_i} c_i n_j \right)$. The flow constraints (as presented in the previous section) are reformulated to fit into the form $A\vec{x} + B\vec{z} + \vec{c} \geq \vec{0}$

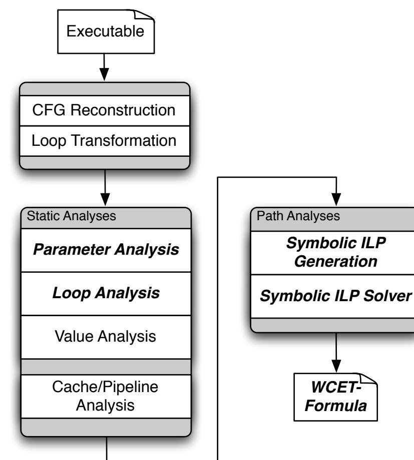

Figure 3. Structure of the parametric timing analysis

Symbolic integer linear programming [8] has been developed by Paul Feautrier and has already been proposed for timing analysis [14] – but within a different context. As the name indicates, integer linear programming is only able to handle linear constraints. A parametric analysis, however, shifts the evaluation of the parametric expressions out

[1] Since programs spend most of their running-time in loops, parametric loops are of main interest.

of the analysis to the instantiation of the symbolic formula. By this, the analysis exhibits two inherent sources of non-linearity: first, not all loop bounds can be described by a linear constraint. Consider Figure 4 where the loop in procedure $f1$ is bounded by p^2 and the loop in procedure $f2$ is only bounded if the parameter p is positive. The second source of non-linearity is caused by nested loops.[2] In Figure 5, the inner loop is bounded by the parametric expression $p \cdot q$ which obviously can not be described using a linear constraint. Both sources are solved by introducing so-called *artificial parameters* which are used to hide the non-linearity from the symbolic ILP.

```
proc f1(p)
begin
  i := 1;
  while (p*p>i) (
   ....
   i := i + 1;
   )
end
```

```
proc f2(p)
begin
  i := 0;
  while (p!= i) (
   ....
   i := i + 1;
   )
end
```

Figure 4. Two loops with non-linear parametric loop bounds.

```
proc f3(p,q)
begin
  i := 0;
  while (p != i) (
     ....
     while (q != j) (
        ...
      )
   )
end
```

Figure 5. Nested loop with non-linear parametric loop bound

In the remainder of this section, we describe the parts of the parametric analysis in detail.

3.1 Parameter Analysis

On the executable level, there are only registers and memory cells and no classification into parameter or variable. In our context, each register and memory cell used within a program is a variable. A parameter is simply a variable the program reads from before it writes to. This is because the value of such a variable before program execution may influence the program flow. However, for the subsequent loop analysis, computing the set of parameters at a certain program point is not sufficient. Instead, we have to compute at each program point a set of so-called *parameter dependencies*.

[2]Although the analysis works on assembler level, some examples are given in pseudo code for the sake of simplicity.

Definition 1 (Parameter Dependency)
A parameter dependency D is a triple $(v, p, [a, b])$ consisting of a variable v, a parameter p and an interval $[a, b]$ where $a, b \in \mathbb{Z}$.

The interpretation is the following: if at a given program point a dependency $D = (v, p, [a, b])$ holds, then also the following holds:

$$v = p + c \text{ where } c \in [a, b]$$

Parameters are usually stored in memory whereas processors operate on registers. If the analysis stores only the set of parameters without the relation to the registers, it loses important information and is unable to compute parametric loop bounds in most cases. The additive interval is also used to include more cases, namely those where the variables linearly depend on the parameter.

The purpose of the parameter analysis is to compute the dependencies defined above. For this, we use the value analysis as in the numeric timing analysis which computes the values for the above intervals.

The parameter analysis is a *data flow analysis* using *abstract interpretation* [4]. The instructions of the program are analysed in an abstract fashion. In Figure 6, for instance, if the program loads the content from memory cell $0x42$ to register $R4$ and memory cell $0x42$ has not been written to before. The analysis considers the memory cell a parameter and adds the dependency $R42 = \mathit{Mem}(0x42)$ to the current program point. When register $R4$ is increased by 10, the dependency is changed to $R4 = \mathit{Mem}(0x42) + 10$ and when the register is loaded with a constant, the dependency does not hold any longer.

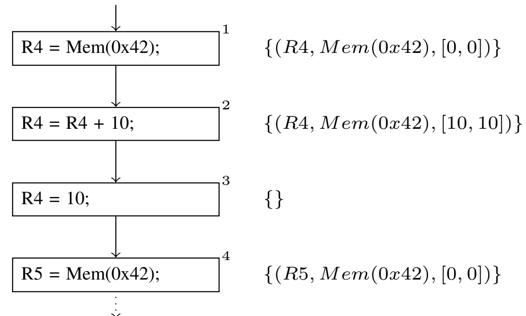

Figure 6. Parameter analysis

Note that in the current implementation, the analysis can only handle dependencies with constant offsets. This means that dependencies of the form $v := 2p$ or $v := p + q$ where

p and q are parameters can not be derived and the analysis relies on user annotations. In most cases, however, dependencies with constant offsets are sufficient; usually, the parameters are used directly without changing their value beforehand.

3.2 Parametric Loop Analysis

The purpose of the parametric loop bound analysis is to compute the loop bounds for each parametric and non-parametric loop. Obviously, the latter problem can be solved as in numeric timing analysis. Inputs to this step are the parameter dependencies for each program point as computed in the preceding step and the values in registers and memory cells (as far as the value analysis was able to derive them).

The numeric loop analysis derives loop bounds by collecting loop invariants. For all potential loop counters, it computes how much they are changed during one iteration, determines initial values for these variables, evaluates the loop exits, and finally constructs the loop bounds. Therefore, the loop analysis consists of four phases:

1. Collection of (potential) *loop counters*.
2. Derivation of *loop invariant*.
3. Evaluation of loop exits.
4. Construction of loop bounds.

All variables changed within the loop body are potential *loop counters* - the first phase collects these variables. The second phase determines the *loop invariant* for the potential loop counters. Such loop invariants have the form: "loop counter i is incremented in each iteration by at least 2 and at most 3". The third phase then evaluates the loop exits. This means, the analysis determines the values the loop counters are compared with at the loop exit conditions. In the last phase, the collected information is combined to compute bounds on the number of loop iterations.

The numeric analysis uses for all of these steps the intervals computed by the value analysis and derives loop bound intervals $[a, b]$ where $a \in \mathbb{N}$ denotes the lower bound and $b \in \mathbb{N} \cup \{\infty\}$ the upper bound.

In the parametric case, however, loops are not bounded by intervals but by symbolic expressions. Therefore, we extended the above method by symbolic evaluation. The parametric analysis not only acquires intervals from the value analysis but also parameter dependencies derived by the parameter analysis. This information is sufficient to compute safe loop bound expressions. The evaluation of the loop exits as well as the construction of the loop bounds has been replaced by symbolic evaluation.

Consider the loop given in Figure 7. Assume that register $R4$ depends on the parameter in memory cell $0x42$ ($R4 = Mem(0x42)$ or, in terms of parameter dependencies as computed in the preceding step, $(R4, Mem(0x42), [0, 0])$) holds.

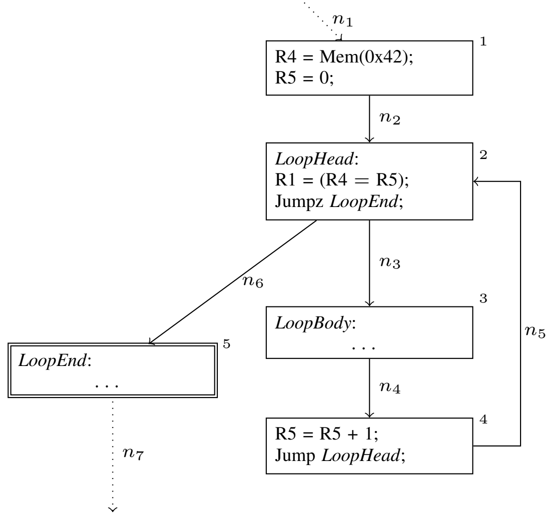

Figure 7. Simple loop example for the loop analysis

The register $R5$ is a potential loop counter (since it is accessed within the loop body). Its initial value is 0 and it is incremented by 1 in each iteration. The loop exit compares the register $R5$ with the register $R4$. Since this register is parametric, the value analysis could not compute a bounded interval for it. The parameter analysis, however, derives the dependency $R4 = Mem(0x42)$. At this point, the analysis has collected all information needed to derive a symbolic loop bound expression:

$$\text{if } \mathit{Mem}(0x42) < 0 \text{ then } \infty \text{ else } \mathit{Mem}(0x42)$$

The evaluation of this expression has to be postponed until the value for the parameter at memory cell $Mem(0x42)$ is available.

As the parameter analysis, the parametric loop bound analysis uses the framework of abstract interpretation [4]. The collection of the potential loop counters and the invariant derivation are data flow analyses that interpret the program in an abstract fashion.

Note that the loop bounds, derived by the loop analysis, are always relative to the loop entry edge. This means that if a loop is bounded by an expression E, then the loop is executed E times the number the loop is entered.

3.3 Pipeline Analysis

In modern processors, the execution time of a single instruction can not be considered constant - it highly depends on the previously executed instructions. Not only the cache

behaviour but also more complex features such as out-of-order execution, branch prediction, and speculative execution can lead to varying execution times for a single instruction. A miss-predicted branch followed by a speculative execution, for instance, causes the processor to spend time in a dead path and to waste time compared to a correctly predicted branch. Thus, the low-level behaviour of the processor must be taken into account to derive safe and tight upper bounds on the execution time.

Although the pipeline analysis is not changed with respect to numeric timing analysis, it is a crucial part of the parametric timing analysis: it simulates the low-level behaviour of the target processor, i.e. the behaviour of the pipeline and the cache, in an abstract fashion [20, 10, 13].

The pipeline analysis is based on an abstract processor model that abstracts away from all features not relevant to the timing behaviour. For instance, the execution time of an arithmetic operation does not depend on the actual operands but on the occupied execution unit. The same holds for a memory access: not the accessed data but the address of the access influences the latency and thus the timing. Caches, branch prediction, prefetch queues and so on, however, are still part of the model. With this abstract processor, the pipeline analysis simulates the execution of the program along the control flow graph and derives upper bounds on the execution times of the basic blocks. It hereby overapproximates all possible execution traces through the processor. In case the control flow splits, the abstract processor states are forwarded to all possible successors and in case the control flow converges, the abstract processors states of all predecessors are combined.

Due to the fact that the abstract model contains all timing relevant features and overapproximates all possible hardware traces, the analysis computes safe upper bounds and even takes timing anomalies [17] into account.

3.4 Parametric Path Analysis

The parametric path analysis computes the WCET formula by symbolically searching the longest execution path in the program. As in the numeric timing analysis, implicit path enumeration is used to generate flow constraints. These flow constraints, however, must be linear in order to be used in an ILP. Therefore, the usually non-linear loop bound expression that are computed by the loop analysis, are replaced by *artificial parameters*.

Definition 2 (Artificial Parameter)
An artificial parameter AP is a symbolic representation for a loop bound expression. It may contain each type of expression (including constants, parameters, and also artificial parameters) except for traversal counters.[3]

[3] Artificial parameter are also used to represent user-specified loop-bound annotations. In case the analysis could not derive loop-bound expressions, the user can specify the loop as bounded by a parameter. Hereby, an artificial parameter is introduced.

Consider the loop in Figure 7 and its loop bound:

$$\text{if } \mathit{Mem}(42) < 0 \text{ then } \infty \text{ else } \mathit{Mem}(42)$$

This loop bound is represented by an artificial parameter AP_l during the path analysis. The flow constraints for this loop are thus given by

$$n_1 = n_2$$

$$n_2 + n_5 = n_3 + n_6$$

$$n_3 = n_4$$

$$n_4 = n_5$$

$$n_6 = n_7$$

$$n_3 \leq AP_l n_2$$

and the cost function by

$$\max : n_1 c_1 + n_2 c_2 + n_3 c_3 + n_4 c_4 + n_5 c_2 + n_6 c_5$$

where c_i denotes the maximal execution time of the corresponding basic block i.

Although the last constraint contains no loop bound expression, it is still non-linear since a parameter is multiplied by a variable. The problem is caused by the relative loop bounds, which are thus converted to absolute ones. In order to perform this conversion, all variables have to be bounded. In the above example, the variable n_2 is obviously bounded by 1 and thus the absolute loop bound is given by $n_3 \leq AP_l$. In case of nested loops, variables may be bounded by artificial parameters again (see Figure 5 for an example). Assume an inner loop is bounded by a loop bound expression represented by AP_1 and the corresponding outer loop by an expression represented by AP_2. After the conversion to absolute loop bounds, there is a constraint of the form

$$n_x \leq AP_1 AP_2$$

which is again non-linear. Thus, a new artificial parameter $AP_x = AP_1 AP_2$ replaces the product of the other artificial parameters and the new constraint is given by

$$n_x \leq AP_x$$

In general, the analysis first bounds all variables (either by a constant as in the simple case or by an artificial parameter) and then replaces the relative loop bounds by absolute ones. Obviously, if all loops are bounded, all variables are bounded too. Note that we lose information during the conversion from relative to absolute loop bounds; absolute loop bounds are less precise since they do not obey the relationship between loop entry edge and loop bound.

The constraint system after the conversion is completely linear and thus forms a valid symbolic integer linear problem. This ILP is then solved by a symbolic ILP-solver. Note that we use in the implementation *PIP* [7], a freely available solver developed by Paul Feautrier that uses symbolic versions of the simplex [6] and cutting plane algorithm [11].

3.5 Instantiation

The result of the parametric path analysis is a conditional expression in the artificial parameters. Therefore, the analysis provides a last step, namely the *instantiation*. Additionally, since the output of the symbolic ILP-solver is rather complex and hardly human-readable, the instantiation step provides a pretty-printer for the result.

A (pretty-printed) result for one parameter is usually of the form:

$$\text{if } AP_1 > c_l \text{ then } AP_1 c_m + c_n \text{ else } c_o$$

$$AP_1 = \text{if } \textit{Mem}(0x42) < 0 \text{ then } \infty \text{ else } \textit{Mem}(0x42)$$

where c_l, c_m, c_n, and c_o are constants and *Mem(0x42)* is a parameter. In case there are more than one parameter, the formula contains nested conditions in these parameters.[4] The resulting formula and the loop bound expression are usually quite simple and clear - at least after pretty-printing.

For the evaluation, the user provides values for the (non-artificial) parameters, i.e. the values the parametric registers or memory cells hold before program execution starts. These values are then used to evaluate first the artificial parameters and then the timing formula. Note that in case a loop bound for a parametric loop has been annotated by the user directly, the user can also provide a value for this parameter directly.

4 Measurements and Discussion

In this section, we describe the precision of our method: a short theoretical discussion is followed by some practical results.

4.1 Loss of Precision

There are two sources that may lead to a loss of precision compared to the numeric timing analysis. The first one is the less precise detection of infeasible path and the second one is the information lost during the loop bound transformation (relative to absolute).

Parametric timing analysis may be unable to exclude paths that the numeric analysis can exclude. If the numeric analysis is given a loop iteration count or the value of a variable which must be considered parametric otherwise, the analysis can use this information to compute more precise results. This affects on the one hand te complete program path that might depend on a parameter, and on the other hand the low-level analysis. The low-level analysis, for instance, distinguishes between the first and all subsequent iterations of a loop. The numeric analysis can often disregard the cache states after the first run (in case the number of loop iterations is at least two), the parametric analysis has to consider always all cases and thus has a less precise cache- and pipeline-state.

The second reason for a loss of precision is depicted in Figure 8. If the path analysis could handle relative loop bounds, the computed worst-case execution path would contain only one of the loops (L1 or L2), whereas the parametric formula contains both loops: the loop entry edges n_{l1} and n_{l2} are bounded both by 1. The information that only one of the loops and not both are executed during a single run is lost. Although only one of them is actually taken, the parametric worst-case execution time includes both. This may lead to a rough over-approximation.

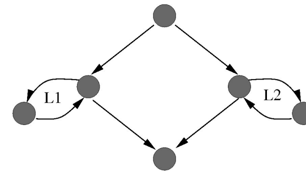

Figure 8. CFG with two loops on Disjoint Paths

4.2 Measurements

Apart from the theoretical description in the last section, we now discuss the practical evaluation of our analysis. We implemented a prototype of our parametric timing analysis for the PowerPC 565 and PowerPC 755. Both are rather complex processors that use a wide span of recent techniques, including out-of-order execution and branch prediction and thus exhibit timing anomalies [15, 17, 20]. Note that the evaluation is based on the timing analysis for the PowerPC 565.

We compared our results against the non-parametric analysis. This means, we compute the symbolic formula once and instantiate it for several values in the parametric case (PA) and in the non-parametric case (NPA), we annotate the loop bounds and start one analysis for each parameter assignment.

Note that the numeric analysis uses the program *lp_solve*, whereas the parametric analysis uses *PIP* for solving the integer linear programs. The tests have been performed on Intel Core Duo 1,66 Mhz with 1024 MB RAM and compiled with a gcc-cross-compiler. The parametric formulas which we provide in the following are direct transcriptions from results obtained by the parametric timing analysis. For the sake of simplicity, we omit the loop bound expression; they mainly denote the actual memory cell that holds the value of the parameter.

[4] In theory, the number of parameters is unbounded, in practise, however, the symbolic ILP-solver PIP is the bottleneck.

We used the following test programs:

- Factorial
- Nested Loop
- Insertion sort
- Matrix Multiplication
- Square Root Computation by Taylor Series
- Finite Impulse Response Filter
- Cyclic Redundancy Check

In all cases except for the last one, we computed the timing bounds for the values $0, 1, 10$, and 100. The cyclic redundancy check is an example containing two parameters. Here, we have chosen $0, 1, 10$, and 100 for the first and $8, 64, 128$, and 256 for the second parameter. The upper bounds on the timing guarantees are given in the number of cycles. Note that the last three test programs are taken from the Mälardalen WCET benchmark suite.[5] Unfortunately, it was not possible to obtain real-life application from the industry to evalutate the method.

Factorial

The first example computes the factorial of n, where n is a parameter. The program has one unique path and must be considered to be rather simple. Table 1 shows the results. As one can see, the parametric analysis gives for all used parameter assignments exactly the same results as the numeric timing analysis.

n	NPA	PA	Diff. in %
0	574	574	0
1	770	770	0
10	2 309	2 309	0
100	17 699	17 699	0

$$\mathit{Time}(n) = \begin{cases} 574 & \text{if } n < 1 \\ 171n + 599 & \text{otherwise} \end{cases}$$

Table 1. Results for factorial

Nested Loop

The next example consists of a triangular loop. The outer loop is bounded by the parameter and the inner loop is bounded by the outer loop's loop counter. The results can be seen in Table 2.

For all values larger than zero, we have a constant over-approximation, which amortises as the parameter grows. The over-approximation is caused by the conversion from relative to absolute loop bounds, where the analysis loses some information about the relationship between inner and outer loop.

n	NPA	PA	Diff in %
0	686	686	0
1	997	1 049	5.2
10	15 686	15 764	0.5
100	1 365 686	1 365 764	<0.1

$$\mathit{Time}(n) = \begin{cases} 686 & \text{if } n < 1 \\ 135n^2 + 150n + 764 & \text{otherwise} \end{cases}$$

Table 2. Results for nested loop example

Insertion Sort

The insertion sort benchmark is a more complex but completely structured program, which contains in our case one normal parametric loop that initialises an array of size n and one parametric nested loop that sorts the values contained in this array by the insertion-sort-algorithm. The results are shown in Table 2.

n	NPA	PA	Diff in %
0	1 494	1 798	20.3
1	1 910	2 086	9.1
10	118 411	121 579	2.7
100	10 788 631	10 791 799	<0.1

$$\mathit{Time}(n) = \begin{cases} 1798 & \text{if } n < 1 \\ 2086 & \text{if } n = 1 \\ 1067n^2 + 1188n + 2999 & \text{otherwise} \end{cases}$$

Table 3. Results for insertion sort

Only for very low values, the parametric analysis is high above the numeric timing analysis. For the value 100 the difference is negligible.

Matrix Multiplication

The fourth benchmark first initialises and then multiplies two matrices of size $n \times n$. It uses the naive approach with nested parametric loops of depth 3. Results are shown in Table 4.

n	NPA	PA	Diff in %
0	2 915	3 046	4.5
1	5 244	8 371	59.6
10	745 156	890 884	19.6
100	669 888 316	683 246 374	2.0

$$\mathit{Time}(n) = \begin{cases} 3046 & \text{if } n < 1 \\ 2431n^3 + 2003n^2 & \\ \quad +663n + 3274 & \text{otherwise} \end{cases}$$

Table 4. Results for matrix multiplication

[5] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

Square Root Computation

This test program computes the square root using Taylor series. The parameter determines the iteration depth and thus the precision of the square root computation.

n	NPA	PA	Diff in %
0	208	208	0
1	208	208	0
10	3331	3331	0
100	34561	34561	0

$$\mathit{Time}(n) = \begin{cases} 208 & \text{if } n < 1 \\ 347(n-1) + 208 & \text{otherwise} \end{cases}$$

Table 5. Results for square root computation

Cyclic Redundancy Check

The last benchmark program performs an 8bit cyclic redundancy check. (Also taken from Mälardalen WCET benchmark suite). The parameter n is the length of the input stream for which the checksum is computed. We introduced an additional parameter a which denotes the size of the allowed alphabet, the input string contains elements of. Here, we performed the analysis for the values $0, 1, 10$, and 100 for n and $8, 64, 128$, and 256 for a which gives a total number of 16 combinations. See Table 6 for the results.

n	a	NPA	PA	Diff in %
0	8	5969	5969	0
0	64	50545	50545	0
0	128	101489	101489	0
0	256	203377	203377	0
1	8	5969	5969	0
1	64	50545	50545	0
1	128	101489	101489	0
1	256	203377	203377	0
10	8	7328	7328	0
10	64	51904	51904	0
10	128	102848	102848	0
10	256	204736	204736	0
100	8	20918	20918	0
100	64	65494	65494	0
100	128	116438	116438	0
100	256	218326	218326	0

$$\mathit{Time}(n,a) = \begin{cases} 397 & \text{if } n < 1 \wedge a < 1 \\ 796(a-1) + 397 & \text{if } n < 1 \wedge a \geq 1 \\ 151(n-1) + 397 & \text{if } n \geq 1 \wedge a < 1 \\ 151(n-1) + 796(a-1) + 397 & \\ & \text{if } n \geq 1 \wedge a \geq 1 \end{cases}$$

Table 6. Results for cyclic redundancy check

Discussion

The results of the benchmark programs show that accuracy of the parametric timing analysis depends on the structure of the analysed program. Nested parametric loops and parametric loops on disjoined paths causes overapproximation (nested loop, insertion sort and matrix multiplication). This overapproximation is due to the conversion from absolute to relative loop bounds. In two cases (nested loop and insertion sort), the overapproximation is constant, which means that the relative difference decreases as the number of loop iterations grows. Already for parameter values of 10 we have a difference of less than 5%. Only for matrix multiplication, the difference grows linearly and therefore does not scale as well. The first and the last two programs do not contain nested loops. Here the parametric timing analysis is as precise as the numeric timing analysis.

5 Related Work

Several research groups addressed timing analysis and also parametric timing analysis.

Vivancos et al. proposed a method able to derive a symbolic WCET formula [21]. In this approach, they use an extended compiler that provides the control flow graph and additional information for the timing analysis. An iterative algorithm computes the WCET of the loops, which is then used to compute a simple formula for the whole program. The paper mainly focuses on the application of parametric timing analysis and thus gives rather coarse information about the analysis itself. In [3] by Coffman et al., the method has been extended to handle nested loops. Since the upper bound on the execution time is computed bottom up and thus, is only locally valid, the analysis can only handle very simple processors without timing anomalies.

Parametric timing analysis on the source code level has been proposed by Chapman et al. [2] and Bernat et al. [1]. Both approaches rely on user annotations, among others, for the parameters affecting the control flow and for the expressions influencing the parametric loop bounds. Using this information, they build symbolic WCET expressions and solve them using an algebra system such as Maple. In addition to the fact that our analysis operates on executables, the main difference to our approach is the required amount of user annotations.

Lisper proposed another parametric timing analysis [14]. He used polyhedral flow analysis for the parametric loop bounds and symbolic ILP for the path analysis. The method can derive WCET formulas automatically without any annotations. Especially the polyhedral flow analysis is able to derive very precise results but at the cost of high complexity. Since the method has not been implemented yet, there exist no practical results so far. In our approach we used the symbolic ILP as proposed by Lisper. The differences to our method are the parameter analysis and the parametric

loop bound analysis replacing the polyhedral analysis. In addition, our analysis operates on the executables in order to take the behaviour of the processor into account.

6 Conclusions

The parametric analysis presented in this paper is able to derive a symbolic WCET formula automatically. A parameter analysis identifies the parameters of the analysed program and computes the dependencies between parameters and variables. The parametric loop analysis derives loop bound expressions for the parameterised loops using these dependencies. At the end, the parametric path analysis constructs a symbolic ILP which is solved to derive the parametric timing formula.

We implemented prototypes of the analysis for the PowerPC 565 and 755. The computed results are very promising and show that the parametric timing analysis is able to compute precise bounds on a task's execution time. If the analysed task contains nested parametric loops, the results from the parametric timing analysis are often only a few percent higher than those obtained by the numeric timing analysis. If the analysed task does not contain nested parametric loops, the results of both analysis are the same.

In future work, the parameter analysis can be extended to handle more complex dependencies and an improved symbolic optimisation method as well as a parametric deadcode analysis promise to increase the precision. Additionally, real-life test-cases from industry are needed to evaluate the practical applicability of the method.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming. Palma (Spain).*, May 2000.

[2] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Syst.*, 11(2):145–171, 1996.

[3] J. Coffman, C. A. Healy, F. Mueller, and D. B. Whalley. Generalizing parametric timing analysis. In *LCTES*, pages 152–154, 2007.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[5] C. Cullmann. Statische Berechnung sicherer Schleifengrenzen auf Maschinencode. Master's thesis, University of Saarland, 2006.

[6] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[7] P. Feautrier. The parametric integer programming's home `http:\\www.piplib.org`.

[8] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[9] C. Ferdinand. Worst case execution time prediction by static program analysis. *International Parallel and Distributed Processing Symposium*, 04:125a, 2004.

[10] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, 1999.

[11] R. E. Gomory. An algorithm for integer solutions to linear programming. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302, New York, 1969. McGraw-Hill.

[12] J. Gustafsson. WCET challenge 2006 - technical report. Technical report, January 2007.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. 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.

[14] B. Lisper. Fully automatic, parametric worst-case execution time analysis. *Third Internation Workshop on Worst-Case Execution Time Analysis*, pages 77–80, July 2003.

[15] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology.

[16] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In K. Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998.

[17] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[18] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Cheju Island, South Korea, 2000.

[19] H. Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, October 2002.

[20] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, University of Saarland, November 2004.

[21] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 88–93, New York, NY, USA, 2001. ACM Press.