In this article, the problem of finding a tight estimate on the worst-case execution time (WCET) of a real-time program is addressed. The analysis is focused on straight-line code (i.e. code without loops and recursive function calls) which is quite commonly found in synthesised code of hard real-time embedded systems. The analysis exploits the very simple structure of these programs, resulting in a considerable processing time improvement compared to general-case analysis techniques. A comprehensive timing analysis system, called the Program Timing Analyser (PTA), covering low-level aspects (on the assembler instruction level) as well as high-level aspects (on the programming language level) is presented. The concepts of PTA are demonstrated with a detailed example. Also some experimental results are given. On one hand the low-level analysis covers all speed-up mechanisms used for modern superscalar processors: pipelining, instruction-level parallelism and caching. It can handle a unified cache as well as separate caches for data and instructions. The pipelined and parallel execution of assembler instructions is analysed a-priori. Also, the analysis predicts whether memory accesses will hit the caches at run-time. On the other hand the high-level analysis addresses the problem of using the results from the low-level to compute the final estimate on the WCET. This is done by a heuristic for searching the longest really executable path in the control flow, i.e. by taking into account functional dependencies between various program parts. The heuristic represents a reasonable trade-off between accuracy and effort. By exploiting the simple structure of the input code, no user-annotations are necessary, resulting in a safe and efficient analysis.

Keywords: worst case execution time, cache memory, pipelined execution, hard real-time, longest executable path.

1. Introduction

Predicting the worst-case execution time (WCET) of a program is an important task in real-time systems engineering. The WCET has to be determined for several reasons, e.g. system configuration, schedulability analysis or assuring response times. In order to guarantee the response time of a program, it is useful to know its WCET, so the cost of the necessary hardware can be minimised. Furthermore, most algorithms for schedulability analysis assume that the maximum execution times of the involved tasks are known [5], [19], [24]. Hence WCET analysis must be performed in order to guarantee that they will always meet their deadline. Especially in a hard real-time environment – i.e. an environment where the missing of a deadline will result in a catastrophic failure – it is important to make safe prognoses about the WCET.

Exact prediction of worst-case execution times is in general impossible, since this would mean solving the undecidable halting problem. Hence the WCET can only be estimated for programs that are known to terminate. Nevertheless, the estimation should be as tight as possible. On one hand the WCET must not be under-
estimated, because this would invalidate subsequent analysis (e.g. schedulability analysis) that are using the estimated WCET values. On the other hand, over-estimation leads to under-utilisation of the hardware, because then the run-time processor load of the system will be much less than predicted.

In the area of embedded real-time systems design, code is often generated by design tools (e.g. [17], [18], [25], [34]) from high-level specifications, based on e.g. differential equations [35], software-circuits [12], statecharts [13] or petri-nets [20]. The structure of the code is simple: it typically contains no loops, no procedure calls and only static variables (according to C-semantics). Thus, the WCET analysis of this kind of synthesised controller code can be done significantly faster and more accurate than in the general case, where nested loops and recursive function calls may occur. This fact is exploited by the Program Timing Analyser PTA, which is presented in this article.

Typically, WCET analysis can be divided into High-Level Analysis (HLA) and Low-Level Analysis (LLA). The HLA examines the program at the source code level. It has mainly to deal with functional dependencies between various program parts, which have to be incorporated when searching for the longest executable path in the control flow. The LLA explores the object code of the program in order to find the precise execution times of sequences of machine instructions. Hence the atomic unit for the HLA is the basic block, i.e. a sequence of statements that are always executed together (cf. Section 2.3), while the atomic unit for the LLA is a machine instruction. This means that LLA has deep knowledge about the timing behaviour of the target processor but is not aware of the global control flow and data dependencies of the program. In contrast, the HLA has knowledge about the program structure but no knowledge about the underlying hardware, thus HLA and LLA complement each other.

Consequently, accurate WCET analysis must comprise both low level analysis (i.e. caching and pipelining analysis) and high level analysis (i.e. longest executable path search LEPS). On one hand, the three aspects LEPS, caching and pipelining cannot be handled independently: The pipeline performance depends on whether an instruction and its operands are in the cache or not. Additionally, the LEPS needs exact worst-case execution times for each basic block (i.e. each node of the control flow graph) of the program. On the other hand, these parts should be only loosely coupled for two reasons: First, modularisation helps managing the complexity of the analysis problem. Second it makes analysis more flexible, e.g. wrt. to changes of the underlying hardware.

As presented in this article, PTA covers all relevant issues in the context of analysing straight-line code. These are:

- **Caching**: In the past years processor speed has increased more rapidly than the speed of memory chips. As a consequence, processors are often stalled waiting for memory accesses. In order to avoid this, caching is introduced: A small amount of recently accessed data is stored (cached) directly in the processor. This data can be accessed much faster than the main memory (typically by a factor of 10 to 100). Blocks of data are moved from main memory to the cache and back as they are needed (cf. Section 2.1).

The difficulty for the WCET analysis is to predict whether a certain memory access will be cached at run-time (i.e. a cache hit) or not (i.e. a cache miss). Since accesses to main memory are much slower than accesses to the cache, it is important to predict the number of cache misses as tight as possible. Pessimistically assuming that every memory access will result in a cache miss is not appropriate, because this leads to extreme overestimation of the WCET.

The cache analysis in PTA exploits data flow analysis principles that are commonly used in compiler construction [1]. In the general case, loops in the control flow graph of a program increase the complexity of the data flow analysis. However, since PTA deals with simple programs that consist only of nested if-then-else constructs, the basic blocks of the program can be processed in topological order. Thus the timing information for each basic block can be collected by inspecting each assembler instruction only once.
• **Pipelining**: Pipelined processors do not execute instructions one after another. They handle instructions overlapped or even completely in parallel. Overlapped execution means that e.g. while one instruction is in progress, the next instruction is already fetched from memory (cf. Section 2.2). Moreover, modern processors typically have multiple execution units (e.g. a floating point unit and an integer unit) that operate independently. Furthermore, superscalar processors can issue multiple instructions concurrently in one single clock cycle. This is also regarded to as instruction level parallelism.

Without pipelining it is easy to predict how long a sequence of instructions will take to execute: simply sum up the individual execution times of each instruction in the sequence. But with pipelined execution the processing time of the sequence depends on the order in which the instructions appear and on how they depend on each other. For example, if one instruction needs the result of the previous one, these two cannot be executed in parallel.

In PTA, pipelining analysis is done by modelling the state of the execution units of the processor for a given sequence of instructions, keeping track of when the units are busy the first and the last time.

• **Longest Executable Path Search (LEPS)**: HLA is usually represented by LEPS, since the final estimate on the WCET is equal to the length of the longest executable path in the control flow. This length highly depends on mutually exclusive control flow branches, maximum loop counts etc. (i.e. functional dependencies between various program parts). Hence the longest executable path is not necessarily equal to the longest structural path whose length would overestimate the WCET. Even in case the program to be analysed is guaranteed to terminate, the problem is of huge complexity: For the exact solution all possible input value combinations would have to be considered. This excludes exact approaches from use when dealing with larger applications.

Hence a heuristic of dealing with the problem is used in PTA. Due to the simple structure of straight-line programs, no user annotations are needed, and a very efficient analysis is assured. The heuristic provides a reasonable trade-off between analysis results and analysis efforts: the results will still be better than purely structure-oriented methods without spending too much time on finding the exact solution. Basically the method is a combination of a path search heuristic (for structural analysis) and symbolic execution (for executability analysis).

This paper is structured as follows: In the next section we describe the basic terminology and basic assumptions that are dealt with throughout this paper. Section 3 provides an overview of related work. In Section 4 the overall structure and the particular concepts of PTA are presented. A detailed example is shown in Section 5 and in Section 6 some experimental results are presented. Finally, Section 7 gives a conclusion and an overview of future work.

### 2. Terminology and Basic Assumptions

In this section the basic terms, concepts and assumptions that are dealt with throughout this paper are illustrated. First, Section 2.1 explains the principles of caches. Section 2.2 deals with the basics of pipelining and instruction level parallelism, and Section 2.3 defines the terminology regarding LEPS.

PTA is embedded in the modified backend of an in-house developed compiler, thus it can take advantage of the data structures and informations that are already gathered during compilation. Throughout the rest of this article, the PowerPC 604 [27] is assumed as the target processor in order to demonstrate the concepts of PTA. Of course, the presented concepts can be applied to other processors too. Moreover, it is assumed that the analysed programs are not interrupted (e.g. by operating system interrupts) during run-time. The problem that interrupts reset caches and pipelines can be handled using the method presented in [7], by adding the overhead for refilling caches and pipelines to the estimated WCET.
2.1. Caches

Caches are used in order to overcome the increasing difference between processor speed and the speed of memory accesses [16]. Caching exploits the benefits of spatial and temporal locality of data accesses: Spatial locality means that a program typically accesses data items that are located closely together in main memory. Hence, when a program accesses a certain memory address, it is likely to access a nearby address in the near future. Consequently, upon the memory access not only the single requested data item but a whole block of data is loaded into the cache (provided that it is not already in the cache). So it is likely that the next data access will be a cache hit. Temporal locality means that a data item that is accessed once is likely to be accessed again in the near future. Thus, the data is loaded into the cache upon the first access, and the following accesses will most likely be cache hits.

We use the term cache line for the amount of memory that is loaded into the cache when a cache-miss occurs. Usually, the size of a cache line is a few bytes (e.g. 32 bytes of data or 8 instructions). A data block is a block of data from main memory that fits into a cache-line.

There are several ways to map data blocks to cache lines: In a direct-mapped cache each data block is assigned exactly one cache line. The opposite is a fully associative cache, where a data block can be loaded into any cache line. A n-way set-associative cache is divided into cache sets of n cache lines each. A data block can be loaded into any of these n cache lines. The cache set to which the data block is mapped is usually determined by the least significant bits of its address. Note, that for n = 1 the cache is direct-mapped, and for n = (no. of cache lines) the cache is fully associative. For conventional processors n ranges between 1 and 8.

Since the cache is much smaller than the main memory, there is usually not enough place for all the data a program needs at run-time. Thus, when a certain data block is loaded, another one must be replaced from the cache. In a direct-mapped cache the replaced block is obviously the one that is currently in the corresponding cache line. In contrast to this, in a set-associative cache a replacement strategy is needed in order to determine which block from the corresponding set to remove. The most commonly used strategies are 'least recently used' (LRU), i.e. choose the block that has not been accessed for the longest time, and 'random', i.e. choose a block randomly.

Caches are furthermore distinguished regarding the handling of writing accesses to memory: In a write-through cache, writing to a certain memory address not only updates the corresponding cache line, but the address in main memory is also updated at the same time. This means that each write access takes as much time as if it was a cache miss. In contrast, with a write-back cache the main memory is only updated when the corresponding data block is replaced from the cache.

Typically, processors nowadays have separate caches for instructions and data. Instruction caches are a bit simpler since for instructions it is sufficient to have read-only access.

PTA is able to analyse arbitrary cache configurations. Only the basic parameters have to be provided: The size of a cache line, the associativity (i.e. n) and the size of the cache. As replacement strategy LRU is assumed, since it is the most commonly used strategy in modern processors.

2.2. Pipelining and instruction level parallelism

The goal of pipelined execution is to increase the throughput, i.e. the number of completed instructions per cycle. This is achieved by dividing instructions into smaller tasks, e.g.:

- Fetch the instruction from memory (resp. instruction cache)
- Fetch the operands of the instruction from memory (resp. data cache)
• Execute the instruction
• Write back the result

Each of these tasks is handled by a dedicated pipeline stage. Each of these stages works autonomous and passes its result to the following stage. Hence the instructions are executed overlapped, e.g. while the operands of instruction i are fetched, instruction i+1 is simultaneously loaded from memory. Moreover, modern processors typically have several parallel execution units. For example separate units for floating point and integer operations.

The stream of instructions in the pipeline can be interrupted due to data hazards or structural hazards: A data hazard occurs when an instruction needs the result of its predecessor. In this case the pipeline is stalled until the predecessor has written back its result so that it can be read. A structural hazard arises when an instruction cannot be executed by its desired execution unit because this unit is still busy handling a previous instruction. Conditional branches are also difficult to handle during WCET analysis: The decision, whether the branch is taken or not (i.e. which is the next instruction), is the result of the execution stage. Hence, there is a high probability that the wrong instruction is in the ‘Fetch’ stage so that the pipeline has to be flushed and execution starts over with the correct instruction.

In a superscalar processor each stage of the pipeline is able to issue several instructions (typically about 2 to 4) concurrently, provided that there are no data dependencies between them.

In order to make tight WCET predictions, PTA considers all of these mechanisms, i.e. structural and data hazards as well as the parallel and overlapped execution of assembler instructions are predicted.

2.3. High Level Analysis

High level analysis explores the structure of a program. This structure is represented by a control flow graph: Each node of this graph is a basic block, which is a sequence of statements that are always executed together. This means that inside a basic block there are no jump statements and no jump targets (labels). An edge from basic block B1 to basic block B2 indicates that the control flow of the program can reach B2 from B1 (e.g. by a jump statement). Hence, the control flow graph describes all possible control flow paths in the program.

The control flow graph is usually generated by the compiler in order to perform various analysis methods and optimizations [1]. One analysis technique that is commonly used by compilers is data flow analysis, which is also used by PTA (cf. Section 4.1).

The compiler which PTA is embedded in performs some peephole optimizations locally for each basic block. Hence the control flow graph is not altered by the optimizations and the results of the LLA can easily be mapped to the control flow graph analysis of PTA. An approach to transform the control flow information through more sophisticated optimising compilers to the LLA is presented in [9].

3. Related Work

Principally, there are two ways to determine the WCET of a program: dynamically, i.e. by measuring the run-time with some sample input data; and statically, i.e. by analysing the program structure. Since the measuring approach [4] is unsafe (one cannot be sure that the considered input data in fact produces the worst-case), most recent research concentrates on the static approach.
Most publications focus on only one or a few of the aspects: LEPS, instruction caching, data caching or pipelining.

Müller [28] developed a technique, called Static Cache Simulation, to statically predict which instructions will be in the instruction cache during program execution. In that approach instructions are classified as always-hit, always-miss, first-miss and first-hit, by analysing the control flow of the program. In [29] the concept is enhanced from direct-mapped to set-associative caches. The Static Cache Simulation only deals with instruction caches, but it is stated in [29], that work is under way to handle data caches as well. Static Cache Simulation can handle nested loops and procedure calls, thus the analysis is more complex than the one used in PTA. Because PTA only has to deal with acyclic control flow graphs, the basic blocks can be processed in topological order, i.e. each block is visited only once. Furthermore, since each instruction of the program is executed at most once, it is sufficient to classify each instruction (and memory accesses) as either a hit or a miss.

A brief overview of the analysis of data caches is given in [6]. Here, among other things, a graph colouring approach similar to that used for register allocation in compiler construction is suggested. That approach tries to group variables based on temporal locality, i.e. variables that are accessed within one basic block are clustered in memory to fit them into one cache block (spatial locality). Its main goal is to be more confident on the estimated number of data cache misses. In contrast to this, PTA does not alter the locations of variables in memory. Pipelining and program path analysis are not addressed in [6].

[23] presents an approach that addresses both pipelining and instruction caching. A program statement is associated not only with a WCET, but with an abstract description of the current status of the pipeline and the instruction cache: the ‘worst-case timing abstraction’ (WCTA). A program path can be analysed by concatenating and pruning the WCTAs of its basic blocks. The handling of data caches is only briefly mentioned in this paper. The WCTAs are comparable to the pipeline modelling used in PTA. However, they are more storage intensive and less accurate than the data structures of PTA.

The prediction of pipeline performance is discussed in [14] and [30]. There the pipeline behaviour is simulated for a given code segment. [30] introduces the ‘pipeline simulator compiler’ psc, which uses a description of a processor to generate a program that simulates the execution of code on this processor. The main shortcoming of these approaches is that the cache performance is not taken into account. So each memory reference must be pessimistically handled as a cache miss. This disadvantage is addressed in [15], where the Static Cache Simulation is combined with pipeline simulation. This approach is comparable to PTA, where cache analysis as well as pipeline analysis are integrated, too. However, [15] does not take into account data caching and instruction level parallelism, as PTA does.

In [21], [31] the program path analysis is modelled by integer linear programming (ILP). Cache performance is also integrated in this ILP-model. Although the program behaviour can easily be modelled by ILP, the analysis is likely to become inefficient for larger applications, since solving an ILP in general takes exponential time. For some of the test-cases the analysis took several hours. However, [21], [31] observed, that for many programs (with up to several hundred lines of code) the ILPs were solved in reasonable time. The path analysis of PTA uses more efficient graph algorithms, so it is also useful for large programs.

Ignoring functional dependencies, and simply searching for the longest structural path in the control flow graph (e.g. [26]), can also result in severe processor underutilisations, which is considerably improved by PTA’s LEPS. More sophisticated analytical methods which try to find the longest executable path usually require user annotations ([33], [32], [8], [22]). These methods typically cope quite well with complexity problems, and deliver very accurate results. Their disadvantage is that the user has to be an expert on the insights of the static timing analysis tool and of the program code. However, PTA’s LEPS does not need any user knowledge about the timing of the real-time program, or vice versa, i.e. it takes into consideration that many of todays real-time programs are automatically synthesised.
The approach presented in [10] tries to provide a user-annotation-free solution. However, like the above mentioned annotation methods, this approach is devoted to a more general context where the input programs can contain more complicated constructs like loops. However, PTA’s LEPS exploits the simple structure of the input format, and hence offers a very efficient heuristic.

4. Program Timing Analysis

The architecture of the PTA system is shown in Fig. 1. The system is embedded in the backend of an in-house developed compiler. Hence PTA is able to use all information that are already generated by the compiler, e.g. the control flow graph and the assembler instructions for all basic blocks.

![Figure 1: PTA System Overview](image)

Given a task in a C-subset (i.e. straight-line code) which is e.g. generated by a design tool, the compiler constructs the control flow graph (CFG) and translates the input into assembler code, which is then further translated to object code. The Cache and Pipeline Analysis uses the CFG and the assembler code for each basic block (i.e. each node of the CFG) to perform data flow analysis in order to predict the data- and instruction-cache performance (Section 4.1). Using this information, the pipeline behaviour for each basic block is predicted (Section 4.2). The results are CFG-labels, representing a description of the pipeline state at the entry and exit of each block. The labelled CFG is used by the Longest Executable Path Search in order to find the longest executable path in the control flow (Section 4.3). The length of a path is computed by concatenating the pipeline representations of its nodes (Section 4.2). Finally, the estimated WCET of the task, i.e. the length of the longest executable path, is returned.

4.1. Cache Analysis

The cache analysis processes each instruction and data reference of the basic blocks of the program in order to predict whether or not it will be in the instruction- and data-cache, respectively. The cache analysis algorithm performs the following data flow analysis, commonly used in compiler construction [1]: For each basic block BB two sets of ‘active objects’ are computed: the set IN for the entry of BB, and the set OUT for the exit of BB. In our case, the ‘active objects’ are the data-blocks that are known to be currently in the cache. For each basic block BB, IN is the intersection of the OUTs of all predecessors of BB. Thus, IN denotes the set of data-blocks that are certainly cached at the beginning of BB, regardless which program path had been executed before. Then, OUT is derived from IN by adding and replacing data-blocks according to the data references in BB.
The cache analysis works independently of the LEPS, i.e. it does not consider functional dependencies in possible program paths. For each basic block, the worst case of all possible paths leading to this block is assumed, regardless whether this case is actually an executable path. On one hand this separation introduces some pessimism in the analysis, since not all potentially available informations are used. On the other hand, the parting helps to reduce the complexity of the overall WCET analysis problem, and allows for a hardware independent HLA.

The cache analysis algorithm is shown in Fig. 2. A 'data reference' is either an instruction fetch or a data access. The function same_set(b, S) returns the set of data blocks from S that map into the same cache-set as b. The cache is n-way set-associative (cf. Section 2.1). IN is empty for the entry basic block, i. e. no data blocks are cached when execution starts.

Within each basic block BB, the cache behaviour is simulated: All data-blocks that are currently cached get a time_stamp which is updated upon each reference (line 9). When a block must be pushed out of the cache, the block with the minimal time_stamp is chosen according to the LRU strategy (lines 18, 19). It cannot be determined how 'old' the data-blocks in IN (BB) are, since IN (BB) is derived from all possible program paths leading to BB. So all data-blocks have the same time_stamp (i.e. 0) at the beginning of BB (lines 4...6). If there are several blocks with the minimal time_stamp, the analysis must assume in a pessimistic way that they are all replaced (see example in Section 5.1).

For each data reference R, b denotes the data-block it belongs to (line 8). SS is the set of data blocks that are currently in the cache set that b belongs to (line 10). Now, the following cases have to be distinguished:

- b ∈ SS (lines 11, 12): This means b was already cached, so R can be marked as a cache-hit.
- b ∉ SS and |SS| < n (lines 13, 14): This means b was not cached and the cache set is not yet full, so no conflicting cache-lines need to be replaced. R is marked as a cache-miss.
- b ∉ SS and |SS| = n (lines 17...19): This means b was not cached and the cache set is full, so there has to be made place for b first. R is marked as a cache-miss. Here the LRU replacement policy is applied: The ‘oldest’ blocks, i. e. the blocks with the minimal time_stamp, are replaced.

```
1: for each basic block BB in topological order do
2:   IN(BB) = \bigcap_{P \in predecessors(BB)} OUT(P)
3:   OUT(BB) = IN(BB)
4:   time = 0
5: for each data block b in OUT(BB) do
6:   time_stamp[b] = 0 /* each block gets the initial time_stamp  0 */
7: for each data reference R in order of appearance in BB do
8:   b = data_block_no(R) /* determine which data block R is in */
9:   time_stamp[b] = time++ /* update b's time_stamp */
10: SS = same_set(b, OUT(BB)) /* determine what is currently in the corresponding cache set */
11: if b ∈ SS then /* R is already in the cache */
12:   mark R as "hit" /* R is not in the cache */
13: else
14:   mark R as "miss"
15:   if |SS| < n then /* the cache set is not yet full */
16:     OUT(BB) = OUT(BB) \ b /* the data block b is loaded into the cache */
17:   else /* the cache set is full */
18:     KILL = {b | time_stamp[b] minimal in SS} /* find the data blocks with the minimal time_stamp */
19:     OUT(BB) = OUT(BB) \ KILL /* all these blocks are removed from the cache */
```

Figure 2: Cache Analysis Algorithm
Note, that the analysis does not assume separate caches for data and instructions. However, if the target processor has separate caches (which is the case for most modern processors), the instruction cache analysis is a bit simpler than the data cache analysis. Since each instruction of the analysed code is executed at most once, it is not necessary to keep track of which cache-lines are replaced upon a conflict. An instruction that has once been in the cache will never be accessed again, due to the straight-line structure of the code.

4.2. Pipeline Analysis

The information about cache-hits and misses that is derived from the cache analysis is further used for the analysis of the pipelining behaviour. The category (i.e. hit or miss) of an instruction determines the time it needs in the Instruction-Fetch stage of the pipeline. If the instruction is a cache-hit, the fetching takes one clock-cycle. Otherwise, fetching takes much longer, depending on the speed of the memory bus. The same holds for memory accesses: data which is in the cache can be accessed within one cycle, but a cache miss takes several additional cycles.

The pipeline analysis of PTA behaves as follows: For each basic block BB of the program an abstract pipeline model is constructed, which describes the state of the pipeline after executing the instructions of BB. The model is derived by successively inserting instructions into the abstract pipeline. The length of a program path (Section 4.3) is then computed by concatenating the abstract pipelines of its basic blocks.

For each type of instruction (e.g. integer add, floating-point divide, etc.), the analysis needs to know, how many clock cycles are needed by each pipeline stage for execution. This information is read from the 'processor description' file (see Fig. 1). In order to describe the state of the pipeline, the following information is stored for each pipeline stage:

- the time it was occupied first and last during the execution of the basic block,
- the number of simultaneously issued instructions at these moments,
- for the instruction-fetch stage: The data-block number of the currently issued instructions.

The data-block number for the fetch-stage is needed because – for the PowerPC 604 – the simultaneously fetched instructions must be in the same cache-line. So the pipeline-modelling has to keep track of the relative addresses of the instructions, in order to determine whether the next inserted instruction can be issued within the same cycle.

Registers are handled like pipeline stages: While an instruction is being executed, its target register is reserved. Thus, subsequent instructions that want to read this register are stalled until the writing instruction is finished.

Fig. 3 gives an example, how two pipeline models P1 and P2 are concatenated. The tables show some of the PowerPC pipeline stages [27]. The table entries depict the number of currently issued instructions at the respective times. At the beginning of P1, for example, each of the stages FETCH, DECODE and DISPATCH issue 2 instructions concurrently at times 1, 2 and 3. At time 4, the stages SCIU (Single Cycle Integer Unit) and MCIU (Multi Cycle Integer Unit) are working in parallel, each processing one instruction.

In Fig. 3b the overlapping of P1 and P2 is illustrated. In this example, the beginning of P2 completely over- laps with the end of P1. The writeback stage handles the last instruction of P1 and the first instruction of P2 in parallel, therefore the corresponding table entry is 2. The head of P1 + P2 is the head of P1, and the tail of P1 + P2 is the tail of P2. The new length is T1 + T2 - overlap. Note, that inserting an instruction into an abstract pipeline P is the same as concatenating P with another pipeline containing only this instruction.
The construction of the pipeline models for the basic blocks is integrated in the cache analysis (see Section 4.1). As soon as an instruction is recognised as a cache hit resp. miss, it is inserted in the abstract pipeline of the current basic block. Thus, the timing information needed for the program path analysis (Section 4.3) is quickly computed by inspecting each instruction of the program exactly once.

4.3. Longest Executable Path Search

The timing information, collected by the cache and pipeline analysis for each basic block of the program, is further used in order to find the longest executable path in the control flow, the length of which is the WCET of the program.

Usually, considering functional dependencies in the control flow graph is needed to obtain sufficiently tight WCET estimates. Every program might contain mutually exclusive parts, as demonstrated by the code segment in Fig. 4.

The obviously longest structural path in this example is the path using the code in lines 1–2–3–4–5–[6,8]–9–10–11–12–13. However, this path is not executable since the code in lines 4 and 11/12 are mutually exclusive due to corresponding if-statements in lines 3 and 10.

Since the number of possible program paths increases exponentially with the number of control flow branches, it is impractical to test all paths for executability. Thus, the high-level analysis of PTA uses a heuristic which works as follows: First, the k longest paths in the control flow graph are sought (e.g. \(k = \text{number of control flow branches in the graph}\)). Then, beginning with the longest one first, each of them is checked for executability. The length of the first executable path found is returned as the WCET of the program. If all k paths are infeasible, the WCET is bounded by the length of the k-th path. During the search the pipeline con-
The check for executability is another computationally intractable problem for the high-level analysis, which we tackle by using symbolic execution. Symbolic execution is a sort of simulated execution of the program often used by optimising compilers. Here it is used to maintain a virtual memory representing current variable values, as far as they are known. It evaluates program statements while traversing along the path under consideration. The symbolic execution will not try to evaluate each statement perfectly: It will only try its best. This method represents a heuristic, since it is started without knowledge about the input data values, and since it never backtracks to check whether a certain condition was indeed true or not. However, values (or at least value ranges) are assigned when approaching certain branches as in if-statements, i.e. for example if the condition for a true branch is ‘E < 0’, and E is unknown for the moment, then it is assumed for the control flow following the branch that E is negative.

5. Detailed Example

In this section we illustrate the concepts of PTA using the sample code segment from Fig. 4. Cache analysis is explained in Section 4.1, Section 4.2 demonstrates the pipeline analysis, and in Section 4.3 LEPS is described. The example is based on the following assumptions:

- all variables are 8 bytes long
- there are separate caches for data and instructions
- the caches are 2-way set associative with 2 sets
- each set holds 2 lines of 16 bytes (2 variables resp. 4 instructions) each
- the mapping of the variables to the cache sets is shown in Fig. 5. The variables a1, u1, u2, a2, a5, a6 fit into cache set 0, and the variables u3, u4, a3, a4 fit into cache set 1
- the replacement strategy is LRU

---

1. a1 = u1 - u2;
2. a2 = u2 - u3;
3. if (a1 < u4)  
   4. a3 = (u2 - u1) / u3;
5. if (a1 != u4)  
   6. a4 = a3;
   7. else
   8. a4 = a5;
9. a6 = u1 - u3;
10. if (a1 > u4)
11. if (a6 != u4)
12. u4 = (u1 - u2) / u3;
13. a6 = u4;

Figure 4: Example Code Segment
5.1. Cache Analysis Example

The control flow graph of the sample program of Fig. 4 is shown in Fig. 6. Also, the IN and OUT sets are depicted for each basic block. Variable accesses that are predicted to be data cache misses are underlined.

![Diagram of control flow graph](image)

**Figure 5**: Mapping of data blocks to cache sets

**Figure 6**: Control Flow Graph of the example of Fig. 4 with INs and OUTs
When the execution starts in B0, the cache is assumed to be empty, hence IN(B0) = ∅. Now the cache is filled upon the data references to u1, u2 and u3, which are all marked as cache misses. At the end of B0 the cache contains the data blocks ((a1, u1), (u2, a2) and (u3, u4)) which is denoted by OUT(B0). The only predecessor of B1 is B0, thus IN(B1) = OUT(B0). IN(B2) is the intersection of OUT(B0) and OUT(B1), since B0 and B1 are the predecessors of B2.

The predicted cache miss upon the reference to a3 in B3 demonstrates that the cache analysis makes pessimistic assumptions about the blocks that have been previously executed: If the control flow is B0–B1–B2–B3, a3 is cached when execution reaches B3. On the other hand, if the control flow is B0–B2–B3, a3 is not cached. Thus the worst case, i.e. a3 is not cached, is assumed.

In Fig. 7 the construction of OUT from IN is demonstrated more detailed for the basic blocks B4 and B7: At the beginning of B4 the cache contains the data blocks ((a1, u1) and (a2, u2) in set 0, and the data block (u3, u4) in set 1. All these blocks have the initial time-stamp 0. Now there is a load access to a5, which is not currently cached. Consequently, the data block (a5, a6), which belongs to cache set 0, has to be loaded. Since set 0 is full at this time, one block must be moved out of the cache first. According to the LRU-strategy, this is the 'oldest' block, i.e. the one with the minimal time-stamp. In this case, both blocks in set 0 have the same time-stamp 0, since the analysis does not know which basic block had been executed before B4. Hence, the analysis must pessimistically assume that both block (a1, u1) and (a2, u2) are removed from the cache. The updated cache contains data block ((a5, a6) with time-stamp 1 in set 0 and data block (u3, u4) with time-stamp 0 in set 1. The access to a5 is marked as a cache-miss. The next memory access is 'store a4', which also results in a cache-miss. The corresponding data block (a3, a4) belongs to cache set 1, which still has one free entry. Thus no block has to be pushed out of the cache. The resulting cache contents with their time-stamps are shown in Fig. 7.

In basic block B7, the situation is rather different: The first data access, 'load u1', results in a cache-hit. So only the time-stamp of data block (a1, u1) in set 0 is updated. The next data access, 'load u2', is a cache-miss, so the data block (a2, u2) must be loaded into cache set 0. Here the removed data block is (a5, a6), since it has the minimal time-stamp 0. The following data accesses, 'load u3' and 'store u4', both are cache-hits, so only the corresponding time-stamp is updated (Fig. 7).

![Figure 7: Predicted Cache History](image-url)
5.2. Pipeline Analysis Example

Fig. 8 shows the generated assembler code of basic block B7 ("u4 = (u1 - u2) / u3;"). For each instruction it is illustrated whether it is predicted as a cache-hit or cache-miss in the data resp. instruction cache. Since a cache set of the instruction cache holds 4 instructions, 4 instructions are always loaded into the cache upon a cache miss. Thus, after each instruction cache-miss, there are always 3 consecutive cache-hits, provided that none of the 4 instructions is a branch.

```
  no.:  instruction:         comment:  instruction cache:  data cache:  cycles:
  0 addis r3, r0, .4u1@ha  address calculation  hit          5
  1 lfd f1, .4u10l(r3)      load u1           hit          7
  2 addis r3, r0, .5u2@ha   address calculation  miss         36
  3 lfd f0, .5u2@l(r3)      load u2           hit          77
  4 fsub f1, f1, f0         f1 := f1 - f0     hit          80
  5 addis r3, r0, .7u3@ha   address calculation  hit          80
  6 lfd f0, .7u3@l(r3)      load u3           miss         111
  7 fdiv f0, f1, f0         f0 := f1 / f0     hit          146
  8 addis r3, r0, .8u4@ha   address calculation  hit          146
  9 stfd f0, .8u4@l(r3)     store u4         hit          148
```

**Figure 8: Assembler Code of B7**

Furthermore, Fig. 8 shows the predicted execution time in cycles. The analysis detects that e.g. instructions 4 and 5 can be executed in parallel. Thus, instruction 5 does not add any additional cycles to the total execution time after instruction 4 (i.e. 80 cycles). The same holds for instructions 7 and 8. A simplified picture of the resulting pipeline representation for B7 is shown in Fig. 9. Here the allocation of some of the resources of the PowerPC is illustrated: The execution units ‘FETCH’, ‘DECODE’, ‘DISPATCH’, ‘SCIU’ (Single Cycle Integer Unit), ‘FPU’ (Floating Point Unit), ‘LSU’ (Load/Store Unit), ‘WRITEBACK’, and the integer register r3 as well as the floating point registers f0 and f1.

```
--- Time:  1  2  3  4  5  6  ...  76  77  78  ...  143 144 145 146 147 148
FETCH   1
DECODE  1
DISPATCH  1
SCIU    1
FPU     1
LSU    1
WRITEBACK  1
r3      1
f0     1
f1     1
```

**Figure 9: Pipeline representation for B7**

5.3. Longest Executable Path Search Example

Fig. 10 shows the four longest paths in the control flow graph including their WCETs. Note that the lengths of the paths are shorter than the sum of the WCETs of their basic blocks, because the overlapping of the corresponding pipelines is considered.
In the following each of the paths is checked for executability, starting with the longest path \(P_0\) until the first executable one is found. The length of that path is then reported to be the WCET. Table 1 shows the history of analysing path \(P_0\). The virtual memory used by the symbolic execution is represented by the values of the variables used (\(a_1, \ldots, a_6, u_1, \ldots, u_4\)). For most of the variables there is no known value before approaching \(B_0\), indicated by entries ‘U’ (= unknown) in row ‘init’. Only for \(u_4\) an input value is known, which is exploited in the following. The rows of the table list the variable setting after the symbolic execution of each basic block.

Due to no knowledge about \(u_1, u_2,\) and \(u_3\), ‘U’ is assigned to \(a_1\) and \(a_2\) during the execution of \(B_0\). In order to access \(B_1\), \(a_1\) must be less than \(u_4\). Since \(u_4\) equals to 2, it follows \(a_1\) must be less than 2 in order to enter \(B_1\), which is expressed by a corresponding entry ‘< 2’ in column ‘a1’, row ‘B0’. In \(B_1\), \(a_3\) is set to ‘U’ due to lack of knowledge about \(u_1, u_2\) and \(u_3\). In order to access \(B_3\), \(a_1\) must be unequal to \(u_4\) which is equal to 2. This constraint is fulfilled by the earlier setting ‘a1 < 2’ which remains unchanged. After the execution of \(B_3\) the if-statement in \(B_5\) is analysed. In order to access \(B_6\), \(a_1\) must be greater than \(u_4\) which is equal to 2. This contradicts the earlier setting ‘a1 < 2’. Hence path \(P_0\) can be considered to be non-executable, and the symbolic execution of \(P_0\) stops.

Figure 10: Four longest paths
Next, the second-longest path (P1) is explored (see Table 2). Until B2 the analysis behaves the same as in the case of P0. In order to access B4, a1 must be equal to u4 which is equal to 2. This contradicts the earlier setting ‘a1 < 2’. Hence path P1 can be considered to be non-executable, and the symbolic execution of P1 stops.

Next, the third-longest path (P2) is explored. Here the situation is the same as for P0: B6 cannot be accessed because this contradicts the earlier setting ‘a1 < 2’. Hence path P2 is also considered to be non-executable.

Finally, the fourth-longest path (P3) is explored (see Table 3). This time no contradiction is detected. Hence path P3 is executable. Its length (929 cycles) is the final estimate on the WCET of the application example.
6. Experimental Results

PTA has been used to analyse several controller applications for mechatronic systems. Some representative input programs are shown in Table 4: A program consisting of straight-line code without any control flow branches (task1307), two medium sized programs with more complex control flow graphs (task47 and a_zsphp6), and one rather large program with many basic blocks (pkw). The programs were run on the PowerPC target architecture and the predicted data (i.e. the number of instruction- and data-cache misses and the WCET) was measured using the performance monitoring facilities of the PowerPC [10]. Careful inspection of the code and the input data of the test programs showed that the monitored execution times were in fact the worst case. The time to analyse the programs was about 2% - 18% of the total compile time, mainly depending on the number of edges in the control flow graph.

For all data items the percentage by which they were over-estimated is given. The number of cache misses was over-estimated up to 14%, and the WCET was over-estimated 4% up to 13%. In addition, a ‘naive’ WCET analysis was performed: All instruction fetches and data accesses were assumed to be cache misses, pipelining was not considered, and functional dependencies were ignored during path search. As shown in table 4, this lead to extreme over-estimation (more than 8 times) of the execution times.

<table>
<thead>
<tr>
<th>program</th>
<th>lines of code</th>
<th>basic blocks</th>
<th>no. of instruction cache misses</th>
<th>no. of data cache misses</th>
<th>WCET (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>measure</td>
<td>overest.</td>
<td>measure</td>
</tr>
<tr>
<td>task1307</td>
<td>4142</td>
<td>1</td>
<td>558</td>
<td>0%</td>
<td>237</td>
</tr>
<tr>
<td>task47</td>
<td>395</td>
<td>43</td>
<td>64</td>
<td>14%</td>
<td>14</td>
</tr>
<tr>
<td>a_zsphp6</td>
<td>5697</td>
<td>393</td>
<td>2310</td>
<td>6%</td>
<td>542</td>
</tr>
<tr>
<td>pkw</td>
<td>11767</td>
<td>881</td>
<td>4971</td>
<td>7%</td>
<td>1222</td>
</tr>
</tbody>
</table>

Table 4: Measurements

7. Conclusion and Future Work

This article presented the Program Timing Analyser PTA, a new timing analysis system for a complete check of straight-line, hard real-time programs. On the low-level, PTA pre-analyses the caching behaviour of program instructions and static data, and also the pipelined execution of the instructions. On the high level the timing information is used to find the longest executable path in the control flow of the program. Throughoutly, PTA exploits the simple structure of synthesised hard real-time applications, which consist of nested if-then-else constructs and have only static variables. Thus, on the low-level the timing information can be derived very fast by inspecting each instruction of the program only once. Furthermore, the cache analysis can handle separate caches as well as a unified cache for data and instructions. The pipeline analysis can deal with superscalar processors. On the high level PTA offers a very efficient heuristic that takes into account functional dependencies without needing user annotations.

PTA is part of the C-LAB Hard Real-Time System (CHaRy [3]). CHaRy is designed to schedule controller applications on a parallel computer. The controller code is generated from higher-level specifications, like differential equations, block diagrams, and software circuits [12]. Currently, PTA’s low-level features generate and analyse PowerPC code. The code generation module is built using a code generator generator, so it can quickly be adapted to other processors. Analysis for SPARC processors is planned in the near future. However, while code generation and cache configuration can easily be parameterised, the pipeline structures of
various processors differ considerably. Thus, due to the rapid evolution of modern processors, a main issue for future work on PTA is to make the approach more generic, i.e. more easily adaptable to various hardware architectures.

A further improvement of the accuracy of the WCET-prediction might be possible with the following method: After the longest executable path has been found by the LEPS, the caching and pipelining analysis could be performed once again for this path. This would give less pessimism, because the caching behaviour could be simulated more exactly. The set $\text{IN}(\text{BB})$ for a basic block $\text{BB}$ (cf. Section 4.1) would be equal to the set $\text{OUT}$ of the predecessor of $\text{BB}$ on the path, and not the intersection of the $\text{OUT}$s of all predecessors of $\text{BB}$, as in the general case. However, this method is not applied in PTA so far.

Future work also includes the extension with regard to nested loops and procedure calls for both the low-level and high-level analysis. Concerning the low level, this can be done e.g. with the Static Cache Simulation approach [28]. Furthermore, the pipeline modelling should be further refined. For example, out of order execution, speculative execution and dynamic branch prediction – features of modern RISC processors – are not modelled adequately yet. At the high-level appropriate concepts for capturing maximum loop counts are needed (like[11]). Furthermore, a more exact symbolic execution algorithm is under development.

8. Acknowledgements

We thank Hans Hansson and Andreas Ermedahl from the Department of Computer Systems, Uppsala University for helpful suggestions and comments improving the quality of this article. We are also grateful to the members of the department for programming languages and compilers of the University of Paderborn, who provided the compiler which PTA is embedded in.

9. References


