Experimental Evaluation of Code Properties for WCET Analysis *

Antoine Colin Stefan M. Petters
Department of Computer Science
University of York
United Kingdom
{firstname.lastname}@cs.york.ac.uk

Abstract

This paper presents a quantification of the timing effects that advanced processor features like data and instruction cache, pipelines, branch prediction units and out-of-order execution units have on the worst-case execution time (WCET) of programs. These features are present in processors (e.g. PowerPC) that are being widely used in embedded and real-time systems. We present an experimental evaluation of the execution time of a series of synthetic benchmarks and real-life case studies. The execution time is evaluated using extensive testing and a simple WCET technique. We show that the most important factor in reduction of execution time is cache size (both instruction and data cache). Other factors like branch prediction and out-or-order execution have minimal improvements that is cancelled out by the pessimism of the analysis. We also argue that some of the performance gain of advanced processor features also applies to the worst case and although WCET estimates may be more pessimistic the overall impact is that they result in lower WCET estimates.

1. Introduction

There is an increasing trend of using advanced CPU's in modern real-time embedded systems. Although the market for simple 8 bit and 16 bit processors is still very large the needs for additional functionality require the use of more advanced processors. Application fields of such processors are, for example, pilot support and X-by-wire systems. The main features of these processors include: data and instruction cache, branch prediction units, pipelines, multiple execution units and in more advanced processors out-of-order execution (cf. table 1). Although it is known that the performance of such processors is very good in the average case, as processor manufacturers optimise for average case execution time, the behaviour in the worst case is very difficult to predict, even more so when the different features interact. Several anomalies have already been reported in the literature where the rare interaction of some features may produce unexpectedly long execution times [8].

The common theme for timing analysis has been to suggest that HW has to be more predictable (in the temporal domain) so that static analysis can be performed. This implies that speculative features like caches and branch prediction should not be used. In this paper we argue that the benefits in performance of such features also apply to the worst case and although WCET estimates may be pessimistic, they are still much lower than less pessimistic estimates that can be achieved with more predictable systems. We support this claim through experimentation of a series of case studies under different processor configurations, using a very simple timing schema to compute the WCET.

There is an extensive literature on the performance improvements due to these processor features for the average case, however very little has been published on a similar analysis for the worst case behaviour. The most similar work is [7] where the impacts of several hardware features for the MIPS processor are analysed. They focus their work on how tight are timing analysis techniques to model particular features like instruction cache, pipeline, etc. We use a different approach by modelling the actual impact of these features on the actual execution time (not only the pessimism of the analysis). The conclusion they reach is that overestimation factors can be very large and that the main overestimation factor is pipelines if cache latency is small and instruction cache if cache latency is large. We agree with these conclusion. However, the programs they have analysed are very small and require a very small amount of data. One of the largest programs is just the inversion of a 3x3 matrix. For real-life programs that have much larger data memory demands data cache may be more significant.

In this paper we want to convey the following messages:

- For advanced processors, there is a difference between average case and observed worst case due to the speed up introduced by these processor features. When applications have data dependent memory accesses or paths these differences increase. There is an inherent variability of the execution time of programs.
- Safe estimates of WCET can be obtained by adequate testing and using measurements with adequate analy-

* The work presented in this paper is supported by the *European Union* with grant NextTTA IST-2001-32111

Processors	Frequ. [MHz]	Data Cache [KB]	Instruction Cache [KB]	Other Cache [KB]	Branch Pred.	Out-of-order
Atmel TS68040	≤ 33	4	4	–	no	no
ARM 1020E	≤ 325	32	32	–	no	no
ARM 1136JF-S	≤ 500	16	16	–	dynamic	no
MIPS32 4Kp	≤ 300	≤ 16	≤ 16	–	no	no
MIPS 20Kc	≤ 600	32	32	–	dynamic	no
Infineon Tri Core 2	≤ 600	≤ 128	≤ 128	–	no	no
Motorola MPC 7410	≤ 500	≤ 32	≤ 32	≤ 2048	dynamic global	yes
Intel Pentium	≤ 300	16	16	opt.	dynamic	no

Table 1. Processor Feature Overview

sis techniques.

- The benefits of advanced processor features such as data caches, out of order execution and pipelines result in more pessimistic WCET estimates, but overall in smaller WCET values.

The structure of the rest of the paper is as follows. In the next section we describe the features of the processors we are interested in. We then present in section 3 an overview of related work and our approach on WCET analysis. The principal section of the paper is section 4. We conclude the paper with a discussion of results and some conclusions.

2. Processors and Peripheral Hardware

State of the art and future embedded processors are equipped with a variety of features to enhance average performance. In this description we consider the features available in medium and high performance processors currently used in embedded systems and the ability of modelling these features for the worst case.

One of the first techniques deployed in embedded processors were pipelines. Later on, caches were introduced in embedded systems. There is usually a separation of data and instruction caches in the first level and a unified cache for the other levels of the caching hierarchy. The analysis of the latter leads to a state space explosion, which may be avoided by making pessimistic assumptions in the case of static analysis. Additional in the case of static analysis. Additional problems arise in ring buffer structures, where the actual access to a memory array is limited but may affect more than one cache line.

Along with the caches, branch prediction, which tries to determine the outcome of a branch in order to continue issuing instructions down the pipeline, was introduced. Opposed to static branch prediction the temporal impact of dynamic branch prediction is complicated and estimates are not likely to be tight because of the difficulty of characterising the worst case scenario, specially modelling global branch history elements. The analysis of multiple execution units has similarities with the analysis of pipelines. Out-of-order execution units are the newest feature to be included in high performance embedded processors. While the instructions are decoded and retired in order, the exact sequence of execution within the processor core is neither statically analysable, as it depends usually on non-disclosed algorithms and external states (e.g. the availability of data), nor is it possible to access the internal states of the processor to actually detect in which order the instructions are executed, although some opcodes enforce serialisation of instructions which preserves the execution order for this particular instruction.

The scope of this paper is the evaluation of the impact that some of these features have on the WCET of programs. The range of features and processors we target is summarised in table 1 with some sample processors.

The problem of WCET analysis is further complicated by the fact, that none of the above processors work alone but are closely coupled with their surrounding periphery. Modern PC104 boards, for example, are basically equipped with more or less all the interfaces used in PCs. This includes, for example, video memory, which is embedded in the processor main memory and accessed via DMA. Another example are external busses, which are clocked lower than the processor, which leads to a quantisation affect. These features in combination make the problem of providing an exact model almost intractable.

The cost of a cache miss depends on several factors: memory speed, relative bus speed and processor architecture. Even for the same CPU this may vary. For the range of boards these processors may be used, a cache miss may be in the range of 10 to 40 cycles.

The above description illustrates the complexity that techniques that aim at accurate modelling of HW processor features have. On one hand, it is possible to identify the worst scenario for the different configurations alone, however the worst scenario for multiple features is a much harder problem and possible intractable.

3. WCET Analysis

There are two main approaches for computing the WCET of programs. On one hand, measurement techniques determine the WCET of a program by observation of the execution time over a testing period. On the other hand, static analysis techniques build a model of the timing behaviour of the processor and together with an analysis of the structure of the code, they are able to provide an upper bound on the WCET of the code.

3.1. Related Work on Static WCET Analysis

Static approaches split the analysis basically into a low level analysis that determines the execution time of individual basic blocks and a high-level analysis that puts together the results of the low-level analysis to produce a picture of the longest path in the program.

The static analysis is based on a model of the processor and by applying this model to the structure of the code an upper bound on the execution is determined. High-level analysis techniques include tree based approaches [5], path based approaches [12], integer linear programming approaches [15]. In the area of low level analysis we can distinguish between analysis methods focusing on the different acceleration techniques. The cache behaviour is analysed by classifying each memory access as cache miss or hit. Müller et al. have modelled instruction and data caches in [14, 10]. A refined but similar approach is used by Theiling et al. in [13] by classifying each access as *hit, first hit only, miss* or *first miss only*. Symbolic equation systems are used by Blieberger et al. in [3]. Wolf and Ernst use local simulation to obtain obtain bounds on the cache hits and misses in [15]. Both Stappert et al. in [12] as well as Healy et al. in [6] have created very detailed pipelines models considering the overlap of instructions. Recently dynamic branch prediction has been investigated (cf. [5, 9]), but this work was limited to schemes without global memory elements (e.g. global branch history). Furthermore other more advanced features like out-of-order execution and most importantly the interaction of such features is still an open problem.

The advantage of these approaches is that the structural analysis is able to determine safe estimates of the worst case path. However, the main problems are the difficulty of determining the worst case interaction of these features and the construction of the model of the processor which is error prone, time consuming and needs to be repeated for each new processor.

3.2. Measurement based WCET Analysis

Measurement based approaches are mainly used in industry. Usually end-to-end measurements are performed and the longest execution time observed multiplied with a safety factor is assumed as WCET. The main problem is that there is no assurance that the WCET has been captured by the testing process. The worst case may only happen when particular series of rare conditions happen during the same execution. It may be possible to generate test cases that produce these rare cases in isolation, however it is much more difficult to generate the worst combination of these events. As exhaustive testing is usually infeasible, there is a risk of the assumed WCET to be less than the real WCET. However, observation is the best mechanism to really capture what can actually happen. The motto of testing is that ”the best model of a system is the system itself”.

The measurement based approach that we propose, called pWCET, combines the best features of static techniques with measurement approaches. The task of the low-level analysis is to determine the WCET of individual basic blocks. The approach taken by pWCET is to determine their execution time by *measurement* instead from a model of the processor. This has some implications. First programs must be tested in a variety of conditions so that each basic block is exposed to the worst interference from neighbouring blocks. Secondly, in a real system, the measurement process is in most cases intrusive, which will in turn add an overhead of the WCET. The work in [11] provides an overview of different measurement techniques and their implications on the quality and possible use of the obtained data.

Within this paper we have used a cycle accurate processor simulator to perform the measurements. This is ideal, as it allows to switch different architectural features of the processor on and off between the experiment series, while the remaining features are unchanged. Only this allows the comparison of the results obtained. This approach has the same drawbacks of using a model of the processor in static timing analysis in the sense that the estimates correspond to the model of the processor, which could be different from the real processor. It is difficult to verify that the timings of the simulator match with the timings of a real processor. However, this is not the issue of this paper, we are interested in the quantification of the impact of hardware features in a particular architecture.

The approach used in the analysis is summarised as follows, for a full description of the pWCET tool-set see [2]. Each function under analysis is wrapped into a test harness. This is just a piece of code that will call the function we want to analyse under different input conditions (e.g. with different input parameters). This is similar to techniques used in reliability analysis for functional testing.

Each run of the program is performed by the processor simulator. We use a modified version of the SimpleScalar[1] cycle accurate processor simulator for the MIPS processor. One of the good things of the SimpleScalar is that it allows the configuration of the processor with command line arguments. It is possible to configure cache sizes and cache configurations, cache replacement policies, branch prediction (types and sizes), Out of order execution among others. The simulator provides an excellent controlled environment to test the impact of particular changes in the processor architecture. The result of the simulator is a cycle accurate trace of the execution. This trace holds the state of the pipeline at each cycle. We use only a small subset of this trace that holds the time instant when the first instruction of any basic block enters the pipeline. With this information is therefore possible to determine the exact path the pro-

1 `http://www.simplescalar.com`

gram followed as well as the exact time instant when each basic block was started.

For the set of experiments we have performed in this paper we have used a very simple approach for the WCET analysis, which is based on determining the absolute maximum execution time for each basic block. That is, a single integer that describes the longest time it took to run a particular basic block. The question still arises as what is the execution time of a basic block. The standard notion is to define the execution time as the time difference between the time instance when the first instruction of the basic block is started and the time instant when the last instruction of the basic block is completed. This is fine for simple processors but for architectures with pipelines where the execution of basic blocks overlap (the first instruction of the next basic block will start before the last instruction of the previous block has finished) requires the identification of the overlap factor between blocks. This complicates unnecessarily the analysis.

Our approach is much simpler and is based on defining the execution time of a basic block as the difference between the time instant when the first instruction of the basic block is fetched, and the time instant when the first instruction of any of the successors of the basic block is fetched. Although we call this measure the execution time of the basic block the boundary is not clear. Instructions of one basic block run after the basic block has notionally completed. We are not interested in the precise identification of each individual basic block but of the worst path. The definition of the worst case execution time of a basic block captures the interference a basic block suffers from its predecessors (as the pipeline and other CPU resources may be busy). This is actually an advantage as we want to quantify such effects but without having to model them precisely. In essence the measurement approach, given proper testing is able to determine the *effects* of the hardware features without having to model them.

The same definition applies also for the advanced CPU configurations used in the experiments with some modifications. For the case of branch prediction, one can not measure the execution time of a block as the execution time of *any* of the successors in case the branch prediction miss-predicts a branch. When a branch is hit, the branch prediction unit decides whether to consider the branch as taken or not taken, while waiting for the outcome instructions of the selected path are fed into the pipeline. If the prediction was right no pipeline stalls have happened and a significant improvement in execution time is gained. However if the branch is miss-predicted then the pipeline is flushed and the execution starts from the other branch. In this case, the WCET of a block is the time difference between the first instruction of the basic block and the first instruction of the *correct* branch (not the first one that appears on the execution trace). This is easy to implement in the processor simulator by removing from the trace the miss-predicted instructions before the analysis process. The net effect is that after some branches there will be a pipeline stall and on other branches no pipeline stall would happen.

For out-of-order execution the boundary between basic blocks becomes very fuzzy. As instructions can be executed in a different order than the one described in the program, it may be the case that the instructions that determine the boundary of the basic blocks are executed earlier or later. This results in measures of the execution time of the basic blocks that have more variability. However, as we take the maximum ever observed execution time between two blocks this method produces conservative estimates of the execution time of the individual execution times. This effect is the main source of the overestimation in the WCET of the examples later in the paper.

The exposure of basic blocks to the worst case interference from neighbouring blocks rises the question on the amount of testing necessary, to achieve this. Experience shows that the number of different execution times of an individual basic block is small. Depending on the size and construct of the basic block and its surroundings one observes usually between 1 to 5 different execution times for any given block even on up-to-date modern hardware. In some rare cases more then 10 different times are observed. This obviously excludes blocking instructions accessing external hardware like interrupt acknowledgement on PC architectures. Such instructions have to be handled separately, but pose no problem, as they occur rarely in application code. The initially counter intuitive small numbers of different execution times stem from the fact, that within a basic block only a small number of relevant operations are performed. Most local variables will be in cache as the memory addresses on the stack are quite frequently accessed. The instruction cache access pattern is usually quite regular. A basic block per definition contains only one branching instruction at the very end, thus a potential miss prediction penalty occurs only once. The remaining variability occurs due to pipelining effects and potential out-of-order execution. However, even those are kept in bay by cache misses, which lead to a clean up (draining of the execution pipeline) and thus less chaotic behaviour than one might expect.

It is impossible to provide a fixed number of executions with varying input data to assume *enough* coverage of the data. However, it is reasonable to analyse the number of different execution times of all basic blocks during testing. When this number does not grow for a large number of test runs, one can assume sufficient coverage. This relies on good testing practice for choosing the variation in the input data. In our tests we have used between 500 and 10000 executions of the overall programmes. These number may seem on first sight rather small, but as the program spends 90% of its time in 10% its code. The relevant code is executed in the order of millions of times. The code executed less, is subject to more cache misses and thus to less variability of the execution time.

By considering the worst observed value ever we produce a safe estimate of the WCET of each basic block provided that proper testing is performed, even exhaustive testing. One might argue that in a pathological case, the worst case execution time of a basic block is not observed during testing as it consists of a precise combination of path and data structures to exhibit this behaviour and therefore exhaustive testing is mandatory. We have checked our examples for such pathological cases and are confident that they contain no such case. This is further supported by the fact, that a basic block has usually a very limited number of different execution times produced by e.g. cache hits/misses, correct/incorrect branch prediction or state of the pipeline. We do not claim that this approach produces tight estimates of the WCET, only, that it produces safe values because they include the maximum ever observed under extensive testing cases and therefore enable us to perform simple WCET estimates.

The high level approach we use in this paper is the simplest timing schema (cf. [5]). A timing schema is a set of rules that compute the WCET of a program by a post-order evaluation the WCET the nodes of the syntax tree. For instance, a simple timing schema is:

- $W(\mathsf{A}) = k$, when A is a basic block. k is the maximum observed execution time of the basic block.
- $W(\mathsf{A;B}) = W(A) + W(B)$.
- $W(\text{if E then A else B}) = W(E) + \max(W(A), W(B))$.
- $W(\text{for E loop A end loop}) = W(E) + n(W(E) + W(A))$ where n is the maximum number of iterations of the loop.

The timing schema is adequate for simple processors and would tend to be more pessimistic for advanced features as it considers the worst effect of a basic block without considering its context. Even with this pessimism, the timing schema is good enough as it shows how simple high-level analysis can result in safe estimates of the WCET of a program. The pWCET approach described in [1] and [2] is much more powerful than the one described here and used in the experiments in the sense that it is able to capture not a single value but a probability distribution of the execution time of each block. It also implements a timing schema manipulating probability distributions. For the purpose of this paper we are interested only on the maximum observed value and therefore a simple integer based approach suffices.

4. Experiments

We have selected five applications for the analysis. They range from simple programs like the bubble sort or a runge-kutta to complex real programs used in commercial applications like the canny edge detection[2]. Table 2 summarises the features of the examples.

- Bubble sort is a simple sorting algorithm. The main feature is that the worst case path depends on the relative ordering of the input data. The maximum number of swap operations happens when the input data is sorted in reverse order. Note that this may not be the worst case path as it may depend on cache misses. Each run of the experiment consists on the sorting of a randomly generated array of 12 integers. We also use bubble-sort to evaluate the number of experiments that need to be made to have a given level of confidence that the WCET has been found.
- The bezier algorithm draws smooth curves into an image. The main feature of this algorithm is a fixed number of loop iterations, but with arbitrary memory references within the image. Each run of the bezier experiment consists on the creation of 20 bezier lines for 4 random reference points on an 800 by 600 pixels image.
- The third case study is the Canny filter, a standard edge detection algorithm used in video motion tracking systems to locate objects in an image. Each run of the canny experiment consists on the extraction of edges of a 64 by 64 pixel randomly generated image (using the bezier algorithm) with some random noise. The algorithm is quite memory intensive and the actual edge following algorithm does arbitrary memory accesses, which are more diverse, than with the bezier example.
- The fourth case study is the simulation of an inverted pendulum using the runge-kutta algorithm and a fuzzy controller. Runge-kutta is commonly used algorithm for numerically solving differential equations and in the simulation of physical systems. Each run of the experiment consists on the simulation of the controlled system (inverted pendulum) and controller (fuzzy controller) for 100 ms of simulated time (100 steps).
- The final case study is the DES encryption program [3]. Each run of the algorithm performs the encryption of a randomly generated word of 64 bits with a key of 64 bits.

4.1. Experiments Setup

The experiments presented here have been conducted using the pWCET tool-set [2], which relies on the SimpleScalar tool-set [4] to obtain the execution traces.

The SimpleScalar simulator is a flexible and cycle accurate simulator that implements a close derivative of the MIPS-IV Instruction Set Architecture (ISA). More precisely, the SimpleScalar instruction set (also called the PISA, or "Portable Instruction Set Architecture") is a superset of MIPS with a few minor differences and additions. SimpleScalar is capable of simulating binary programs and

2 The full code and configuration scripts are available at `http://www.pwcet.com/samples/rtss03/`

3 Taken from `http://www.c-lab.de/home/en/download.html`

Case study	Size	Description
Bubble sort	10 lines	sorting an array of 12 elements
Bezier	20 lines	Drawing 20 lines of 4 reference points on a 800x600 image
Canny_filter	600 lines	Edge detection of an image of 64x64 pixels
Runge-Kutta + fuzzy controller	300 lines	4th order RK simulation and control of an inverted pendulum
NDES	200 lines	Encryption of 64 bit word with a 64 bit key

Table 2. Benchmarks

IDBOP	I-cache –/i/I	D-cache –/d/D	Dyn. Bpred b/B	Out Of Order o/O	Pipeline P
—boP	off	off	stat.	off	on
i–boP	128B	off	stat.	off	on
I–boP	32KB	off	stat.	off	on
idboP	128B	128B	stat.	off	on
IDboP	32KB	32KB	stat.	off	on
idBoP	128B	128B	dyn.	off	on
IDBoP	32KB	32KB	dyn.	off	on
idbOP	128B	128B	stat.	on	on
IDbOP	32KB	32KB	stat.	on	on
IDBOP	32KB	32KB	dyn.	on	on

Table 3. Configurations used in the experiments

performs out-of-order execution of the instructions while simulating the effects of data and instruction caches, branch prediction, etc.

We use the configuration options of the SimpleScalar to simulate different processors. We analyse each of the programs under 10 different configurations determined by enabling or disabling several features. In particular, we consider enabling or disabling : (i) the instruction and data caches, (ii) the dynamic branch prediction mechanism and (iii) the out-of-order execution. Pipelines can not be disabled.

The 10 configurations are listed in table 3. These range from all features disabled (except pipelines) to all features enabled. We have selected three configurations of caches: no cache (–), small data (d) and instruction (i) cache and large data (D) and instruction (I) cache (cache sizes are defined relative to the program size, small meaning "not all data /code fits in the cache", large meaning "all data/code fits in the cache"). We also consider enabling dynamic branch prediction (B) or using a simple static branch prediction (b), and enabling (O) or disabling (o) out-of-order execution. Not all combinations may be found in real hardware, for example out-of-order execution without branch prediction.

All programs are tested for each of the configurations for exactly the same set of input data. Bubblesort is tested 10000 times, the canny filer for 500 cases, the rest of examples are analyse for 5000 tests. All tests are randomly generated. The experiments are performed assuming that cache size is either 128 bytes or 32Kb, with a cache block of 4 bytes. A cache hit costs 1 cycle and an access to the main memory costs 18 cycles (default value in SimpleScalar). The simple static branch prediction scheme assumes all branches falling through. In this case the pipeline is flushed, whenever the branch is taken. The dynamic branch prediction scheme consists of a two level branch prediction schema. The branch predictor state machines are addressed using a 12 bit branch history register. Additionally the branch predictor unit contains a 512 entry 4 way set associative branch target buffer, which is used by the fetch unit to continue the execution on a branch predicted taken. For each of the experiments we compute some data on the distribution of the observed execution times and the WCET estimates. Given a code section S:

- C^p: is the p percentile of the observed execution time of S, meaning that the probability that a particular execution time is smaller of equal to t is p ($P[t \leq C^p] = p$). We pay special attention to $C^{0.5}$ (the median), $C^{0.9}$, $C^{0.99}$ and $C^{0.999}$. In particular C^1 is the maximum observed execution time of S and C^0 is the best case observed execution time.
- $\hat{C}$ is the *real* worst case execution time. This quantity is possibly non computable.
- C^W is an estimate of the worst-case execution time analysis provided by the pWCET analysis.

One particular property we are interested in modelling is the variability of the execution time. Ideally we would like to be able to reason on how far are estimates (either C^1 or C^W) in relation to $\hat{C}$, however as $\hat{C}$ is not computable, we can only reason about the observed values. We also provide the overestimation of the pWCET approach compared to the worst observed value as $(C^w - C^1)/C^1$. This result may be misleading if C^1 is far from $\hat{C}$.

Note that we can not consider other common statistical measures like the average or the standard deviation as we can not make the assumption that the distribution of execution times follows any known distribution (like the normal). For some of the programs exhaustive testing could be performed and therefore the real worst case execution time, i.e. $\hat{C}$, could be obtained, however, this is unfeasible in the general case.

Although the analysis is performed for the MIPS architecture, the results can be generalised to a family of processors with similar features (although the timing and structure of instructions may be different the same effects occur). The important result is not the absolute values of the timings of the programs but their variation for different configurations.

The purpose of this study is to quantify the impact of advanced processor features for a particular advanced pro-

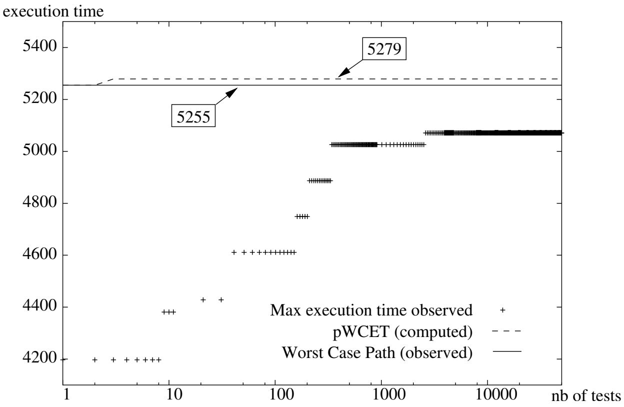

Figure 1. Evolution of the maximum observed execution time (Bubble sort - IDoP) with perfect branch prediction

cessor. We are very careful to claim that this is the temporal behaviour of the MIPS processor. We are aware that the timing information may not match any of a real processor (besides, the instruction set is actually an extended version of the MIPS instruction set). What we can infer from this study is the relative impact that some of the advanced features have relative to each other. On other processor architectures the timings of the individual instructions are different, the architecture is different but we conjecture that the trends are similar.

In addition to the experiments described before, we have also performed an experiment to evaluate how many tests are needed to determine the WCET of a program with the described method. If the input of size n data for the bsort algorithm is randomly generated, then the probability of generating the worst input data is 1 in $n!$. For long arrays this number is intractable. The question to address is how fast do the estimates of worst case converge as a function of the number of tests. This is shown in figure 1. The X axis indicates the number of tests performed and the Y axis the current estimate for WCET. The two lines represent the execution time observed for the worst case path when $n = 11$ and the pWCET estimate. With very few tests pWCET produces an estimate, which is very tight and always above the observed value.

5. Discussion

There are three main significant conclusions that can be drawn from the evaluation of the experiments.

Firstly, the major impact by far among all effects is the cache, both instruction cache and data cache (considering the fact that a cache miss has a penalty of 18 cycles only, the effects would be more dramatic for larger cache penalties). For all experiments, ignoring data an instruction cache (–boP) leads to extremely long execution times compared with the rest of configurations. Having some instruction cache (i-boP) increases dramatically the execution times (WCET included). The next significant increase happens for data cache (idbop). The best results correspond to the configurations with largest cache size (ID??P).

Secondly, the overestimation factor is larger for the more advanced features. For the simplest configuration the WCET is within a few percent of the maximum observed values. For advanced configurations the overestimation factor is more than 50% (WCET is twice the maximum observed value) in most of the examples except for bsort that it is only 36%.

Thirdly, the most surprising bit is that the level of overestimation is much smaller than the loss of performance due to not having the advanced features disabled. For instance, for the bezier example that has 62% overestimation in the most advanced configuration (IDBOP) the WCET is 691807. The *observed best case* for the configuration with medium cache (idbOP) is 1398617, this is twice as long. Even for the bublesort example which has a very strong data dependent execution time, the benefits of the advanced features regarding the WCET are considerable.

Besides these other conclusions can be drawn from the examples. Out of order execution increases the performance of the observed execution time between 15% and 20% when compared to the same configuration without out of order execution. The only exception is the bsort example where out of order execution has only a 1% increase in performance. However, the increased overestimation of the analysis makes this performance not practical for WCET purpose only.

Branch prediction has very little impact in the examples. This may be explained by he short depth of the pipeline of the SimpleScalar mixed with the fact that the default no branch prediction means assuming all branches not taken which actually is a very good predictor.

The programs show a variable execution time. The most notable one is the bsort as the time it takes to sort an array depends on how well sorted is the input data. With a simple processor the worst case corresponds to the input array being sorted in decreasing order. Most importantly, figure 1 shows that the WCET estimate is very close to the real worst case, with only an overestimation of less than 1%. The pWCET analysis reaches this value with only three runs, however the testing approach after more than 50000 runs clearly underestimates it. The same result applies to all experiments. This shows that the pWCET analysis technique requires very few experiments to produce stable estimates of the WCET.

The other case study, which is data dependent is the canny filter. The variation between best case and worst case is quite small (less than 1%). However, all runs analyse very similar images (they contain 20 bezier lines) the variability comes from the different memory access patterns.

The other experiments exhibit much less variability. By analysing the structure of the code, it is clear that the measurement approach may have problems finding the

Bubble sort

IDBOP	BC obs. ET	0.5pct	0.9pct	0.99 pct	0.999 pct	WC obs. ET	pWCET	% overestimation
––boP	37696	54301	59836	63526	66601	71521	74311	3.75 %
i–boP	20862	30360	33475	35590	37340	40126	44111	9.03 %
idboP	15994	22701	24859	26359	27609	29606	31869	7.10 %
idBoP	15117	23016	25202	26728	28025	29065	32914	11.69 %
idbOP	15582	20852	22584	23754	24729	26284	28465	7.66 %
I–boP	13431	22291	25151	27113	28736	31316	33636	6.89 %
IDboP	3557	4828	5196	5442	5647	5975	6492	7.96 %
IDBoP	2648	4246	4639	4920	5161	5399	5950	9.26 %
IDbOP	3415	4594	4898	5128	5326	5622	6492	13.40 %
IDBOP	2632	4235	4629	4912	5139	5350	8354	35.95 %

Bezier

IDBOP	BC obs. ET	0.5pct	0.9pct	0.99 pct	0.999 pct	WC obs. ET	pWCET	% overestimation
––boP	3759873	3884260	3914047	3934771	3941051	3945761	4135345	4.58 %
i–boP	2140277	2190591	2202634	2211016	2213556	2215461	2418840	8.40 %
idboP	1707711	1758355	1770750	1778841	1781978	1784796	2102635	15.11 %
idBoP	1683282	1735563	1748298	1756548	1759877	1762563	2171234	18.82 %
idbOP	1398617	1431686	1439645	1445135	1446782	1448012	1619534	10.59 %
I–boP	1579597	1617613	1626733	1633065	1634965	1636405	1738300	5.86 %
IDboP	323750	330376	331924	333035	333498	333689	589040	43.35 %
IDBoP	300669	309230	311237	312655	313209	313473	600951	47.83 %
IDbOP	268123	273904	275260	276218	276620	276834	589040	53.00 %
IDBOP	251855	260336	262326	263717	264242	264551	691807	61.75 %

Canny filter

IDBOP	BC obs. ET	0.5pct	0.9pct	0.99 pct	0.999 pct	WC obs. ET	pWCET	% overestimation
––boP	11516693	11548779	11565005	11579787	11587797	11587797	12825654	9.65 %
i–boP	6506316	6539903	6554409	6563197	6570674	6570674	7435186	13.15 %
idboP	5394061	5417143	5427515	5436434	5442196	5442196	7160492	23.99 %
idBoP	5259379	5284253	5292702	5300840	5304406	5304406	7224698	26.57 %
idbOP	4573615	4589992	4597387	4605544	4609580	4609580	7160492	35.62 %
I–boP	4653155	4676764	4686431	4695601	4699240	4699240	5200083	9.63 %
IDboP	1057043	1059673	1060920	1061939	1062676	1062676	1442796	26.34 %
IDBoP	932260	937371	938728	939783	940181	940181	1460050	35.60 %
IDbOP	892225	895815	897483	898902	899672	899672	1572900	42.80 %
IDBOP	818664	824190	825657	826834	827353	827353	1799460	54.02 %

Runge-kutta

IDBOP	BC obs. ET	0.5pct	0.9pct	0.99 pct	0.999 pct	WC obs. ET	pWCET	% overestimation
––boP	246150	248114	248114	248114	248114	248114	249954	0.73 %
i–boP	127965	128941	128941	128941	128941	128941	130187	0.96 %
idboP	118402	119222	119222	119222	119222	119222	121188	1.62 %
idBoP	110208	111003	111003	111008	111155	111155	115403	3.68 %
idbOP	104530	105254	105254	105254	105254	105254	107340	1.94 %
I–boP	77307	77851	77851	77851	80937	80937	93691	13.61 %
IDboP	20642	20790	20790	28395	28395	28395	52513	45.92 %
IDBoP	17573	17713	17713	17804	26670	26670	52611	49.30 %
IDbOP	20642	20790	20790	20790	28385	28385	52513	45.94 %
IDBOP	15939	16019	16019	24354	24358	24359	50728	51.98 %

Des

IDBOP	BC obs. ET	0.5pct	0.9pct	0.99 pct	0.999 pct	WC obs. ET	pWCET	% overestimation
––boP	2281376	2281376	2281376	2281376	2281376	2281376	2316276	1.50 %
i–boP	1143321	1143321	1143321	1143321	1143321	1143321	1178245	2.96 %
idboP	1061486	1062523	1062856	1063094	1063243	1063359	1189227	10.58 %
idBoP	1033987	1035007	1035340	1035578	1035726	1035791	1169756	12.93 %
idbOP	682974	682974	682974	682974	682974	682974	886630	22.96 %
I–boP	682974	682974	682974	682974	682974	682974	886630	22.96 %
IDboP	187341	187364	187364	187364	187364	187364	437511	57.17 %
IDBoP	161931	161954	161954	161954	161954	161954	422053	61.62 %
IDbOP	178061	178070	178079	178079	178088	178088	415086	57.09 %
IDBOP	158431	158440	158449	158449	158458	158458	430215	63.16 %

Instruction cache:	– = off	i = 128Bytes	I = 32KB
Data cache:	– = off	d = 128Bytes	D = 32KB
Branch prediction:	b = static fall through	B = dynamic 2 level scheme	
Out of order execution:	o = off	O = on	
Pipeline:	p = off	P = on	

Table 4. Experiment results

WCET for the examples with more variable execution time (bsort and canny). The rest of case studies don't have path dependent execution time and therefore the measurement approaches provide tight estimates. However, it is not clear how much underestimation the measurement only approaches have.

This confirms the idea that advanced processor features are more difficult to model, however the most important result is that even with such levels of overestimation, the resulting WCET estimate is smaller than any observation made with any of the major features disabled. Due to the conservative nature of the WCET analysis it might be even concluded, that only in some pathological cases the real WCET $\hat{C}$ is larger with a particular feature turned on compared to the value with the feature turned off. This result applies to all case studies. This indicates that although advanced processor features are difficult to model and result in variable execution times with more pessimistic estimates

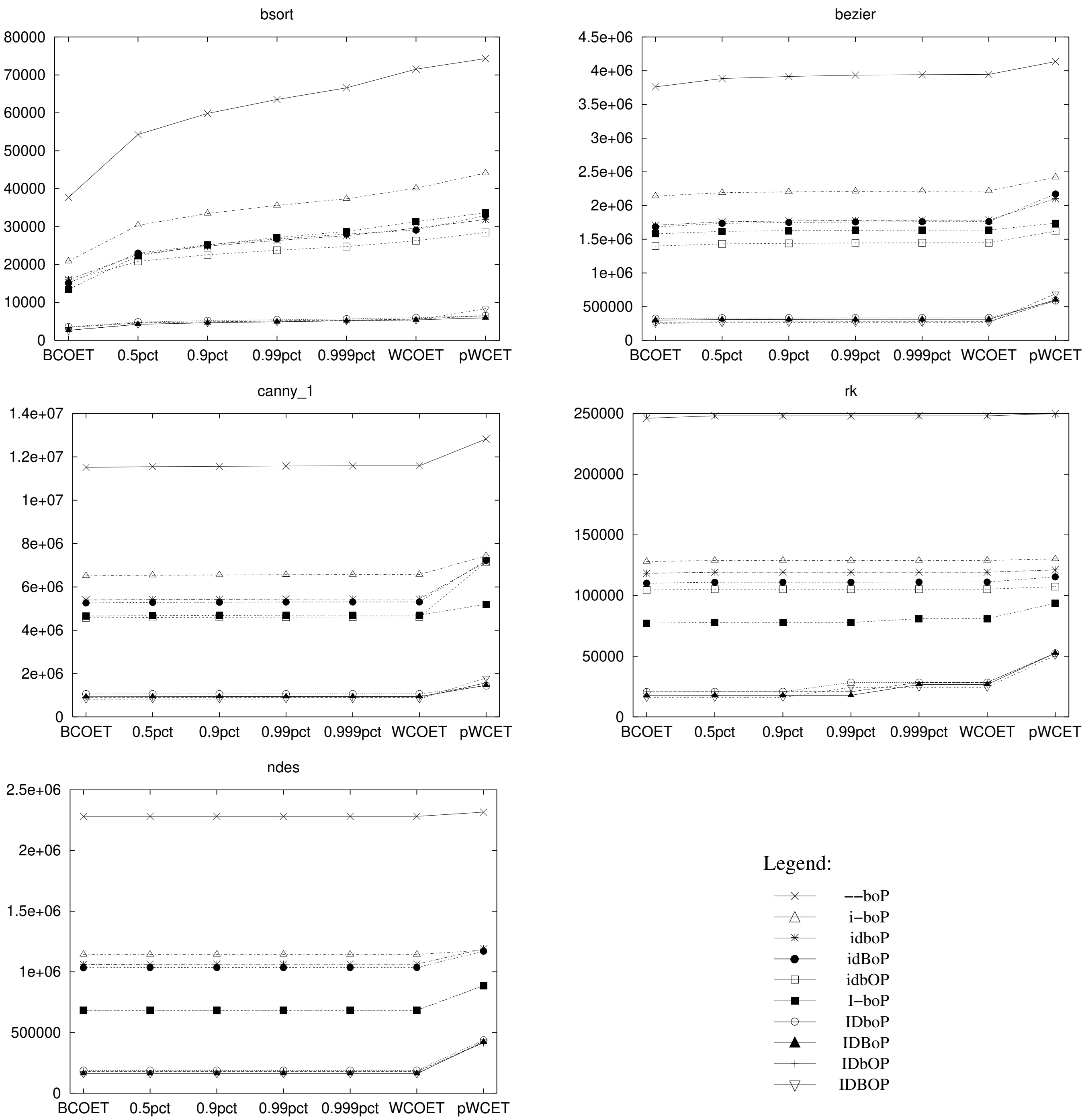

Figure 2. Graphical representation of the data from table 4

of the worst case good estimates of the WCET can be computed. Due to the pessimism in the WCET estimate, tasks would require significantly less execution time than the estimate resulting in gain time being available at run-time.

It is interesting to note that these conclusions apply to all case studies, which are quite varied in size, execution time, path structure and functionality. The results, although produced by the SimpleScalar simulator for the MIPS processor, are generalisable to a wide class or processors with similar hardware features.

6. Conclusion

In this paper we have analysed the worst-case timing behaviour of several case studies ranging from small typical sections of code (bubblesort) to large real-life programs (canny edge detection) running on a simulated MIPS processor under different processor configurations. We have performed the WCET analysis of these programs using a mixture of measurement and static analysis techniques using the pWCET tool-set. Our experiments show that although advanced processor features lead to less predictable WCET behaviour, this is still bounded and results in much smaller WCET estimates that the more predictable system with some of the features disabled. This paper has also shown the effectiveness of the pWCET analysis technique for large programs running on complex processors.

One of the most important features of the pWCET analysis technique is that it does not rely on a model of the processor, only on measurement of actual execution times. This has allowed the integrated analysis of the different processor features (instruction caches, data cache, pipeline, out or order execution and branch prediction) with no additional effort. Moreover, it has been possible to model features for which there are no WCET techniques available like dynamic branch prediction and out-of-order execution. Also, the technique is able to analyse systems where multiple features are present and for which static characterisation of their interaction is not possible. Still, the level of overestimation is high. Further work will involve enhancing the pWCET analysis to provide tighter estimates.

The evaluation of the experiments shows that the feature that accounts for the larger impact in both average execution time and WCET is by far cache size (both data and instruction cache). Our results indicate that this is a very important factor that leads to a significant reduction of the worst-case execution time of programs.

The presence of other hardware features like out of order execution and branch prediction result in modest improvements. However the pessimism of the simple measurement approach results in overestimation factors that brings the WCET to similar levels as the configurations without branch prediction and out of order execution. Future work will involve the reduction of the pessimism in the analysis of such configurations, besides no significant improvements have been achieved.

An important additional conclusion that can be drawn from these experiments is that even if the WCET estimates were tight, there is a wide variability of execution times when advanced processor features are enabled. One can expect the worst case to occur very rarely, even more so taking into consideration the inherent pessimism of the analysis.

The pWCET approach relies on the good quality of the input data. This requires adequate testing strategies so that execution times of measurement blocks have been exposed to the worst interaction from their neighbouring blocks. Emphasis of the definition of test scenarios has to be on this local effects as the worst path and global effects are being captured by the pWCET analysis technique. Further work will involve the analysis of real-hardware with the same experiments described in this paper.

Acknoledgements

The authers wish to thank Guillem Bernat for his strong support in the development of this paper. Further thanks go to Chenyang Lu and the anonymous reviewers for their valuable comments.

References

[1] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real–time systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, Austin, Texas, USA, Dec. 3–5 2002.

[2] G. Bernat, A. Colin, and S. M. Petters. pWCET: a tool for probabilistic worst case execution time analysis of real–time systems. Technical report YCS353 (2003), University of York, Department of Computer Science, York, YO10 5DD, United Kingdom, Apr. 2003.

[3] J. Blieberger, T. Fahringer, and B. Scholz. Symbolic cache analysis for real–time systems. *Journal of Realtime Systems*, 18:181–215, 2000.

[4] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25, June 1997.

[5] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Realtime Systems*, 18:249–274, 2000.

[6] C. Healy, R. D. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *Transactions of computers*, 48:63–70, Jan. 1999.

[7] S.-K. Kim, R. Ha, and S. L. Min. Analysis of the impacts of overestimation sources on the accuracy of worst case timing analysis. In *Proceedings of the IEEE Real–Time Systems Symposium*, Phoenix, AZ, Dec. 1999.

[8] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the IEEE Real–Time Systems Symposium*, Phoenix, AZ, Dec. 1999.

[9] T. Mitra and A. Roychoudhury. A framwork to model branch prediction for WCET analysis. In *2nd Intl. Workshop on Worst Case Execution Time Analysis*, Vienna, Austria, June 18 2002.

[10] F. Müller. Timing analysis for instruction caches. *Journal of Realtime Systems*, 18:217–247, 2000.

[11] S. M. Petters. Comparison of trace generation methods for measurement based WCET analysis. In *3nd Intl. Workshop on Worst Case Execution Time Analysis*, Porto, Portugal, July 1 2003.

[12] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, pages 132–140, Atlanta, GA, Nov. 16–17 2001.

[13] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by spearated cache and path analysis. *Journal of Realtime Systems*, 18:157–179, 2000.

[14] R. White, F. Müller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis of data caches and set–associative caches. In *3rd IEEE Real–Time Technology and Applications Symposium*, Montreal, Canada, June 9–11 1997.

[15] F. Wolf and R. Ernst. Data flow based cache prediction using local simulation. In *Proceedings of High Level Design Validation and Test*, Berkeley, USA, Nov. 2000.