Lehrstuhl für Realzeit–Computersysteme

Worst Case Execution Time Estimation for Advanced Processor Architectures

Dipl.–Ing. univ. Stefan M. Petters

Lehrstuhl für Realzeit–Computersysteme

Worst Case Execution Time Estimation for Advanced Processor Architectures

Dipl.–Ing. univ. Stefan M. E. Petters

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor–Ingenieurs (Dr.–Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.–Prof. Dr.–Ing. K. Antreich

Prüfer der Dissertation:
1. Univ.–Prof. Dr.–Ing. G. Färber
2. Univ.–Prof. Dr. techn. J. Swoboda

Die Dissertation wurde am 14.03.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 22.08.2002 angenommen.

Preface

Holzkirchen, 13th March 2002

As it is always with such a work, it needs the support of many people to be accomplished. First to be named is my supervisor Prof. Georg Färber. He took me under his wings even before I had finished my diploma thesis. His considerable and never slacking interest in the subject of this thesis has bourne me over a time, where the work seemed to become a never ending story. In a similar early stage I got support of Prof. Joachim Swoboda, who has taken upon him the task of doing the second review of my thesis. My collegues at the institute deserve many thanks for proof reading my publications, asking questions regarding my work, which sometimes got me flat footed, sustaining a good working climate and occasionally providing interesting tasks to avoid working on this dissertation. I want to mention explicitely Alexander v. Bülow, who gave me a good deal of thrust in motivation during his diploma thesis.

Muireann Bennis helped to fill in a lot of commas which I only use sparingly, pointed out many Germish errors, and sentences stretching over half a page. Last but definitely not least my wife Amine deserves many a thanks for her continuous support and trust in my work. Besides thanks I have to apologise to my wife and especially my children, Shahin and Anian who had to do more or less without husband and father for quite some time now. I hope I can make up for this time.

Stefan M. Petters.

Often you must turn your stylus to erase,
if you hope to write anything worth a second reading.

-Horace, poet and satirist (65-8 BCE)

Abstract

Advanced acceleration features, as they are used in todays mass market, high performance processors, have only been considered in isolation in previous worst case execution time estimation approaches. This thesis presents a measurement based approach to estimate the worst case execution time on a fully featured processor. To produce reliable results several aspects have to be considered. Prior to the start of a measurement, the acceleration techniques are preset, as far as possible, into their worst case state. The features, which cannot be controlled to produce the worst case state are either randomised or covered by penalties added to the measured results. All possible path combinations are enforced using additional instrumentation code. By partitioning the measurement problem into several measurement blocks, the coverage of all path combinations is ensured. To cover final uncertainty, an existing extreme value statistic approach is extended, to handle combinations of measurements. Additionally a scheduling analysis method, suitable for processors equipped with such acceleration techniques, is presented. A number of test cases, show the applicability and the limitations of the approach.

Contents

Glossary

factual dead code	code which may never be reached, regardless of the input parameters, but cannot be identified as such by a compiler.
B_i	Blocking time suffered by thread τ_i induced by resource contention.
C_i	Uninterrupted WCET of thread τ_i.
$E_j(R_i)$	Worst case number of releases of thread τ_j during time R_i.
p_{AWCET}	Confidence level of an Assumed WCET value t_{AWCET} for a specific piece of code.
$P_{i,j}^n$	Worst Case number of preemptions of thread τ_i by thread τ_j which are not covered at start of iteration n.
R_i	Worst case response time of thread τ_i.
T_j	Minimum interval between two releases of thread τ_j.
t_{AWCET}	Assumed WCET value for a specific piece of code.
t_{MWCET}	Measured WCET value for a specific piece of code including all additional penalties.
t_{WCET}	Physical WCET value for a specific piece of code, which is usually unknown.
w_i^n	Execution window of thread τ_i in the nth iteration.
δ_i	Worst case individual preemption delay suffered by thread τ_i each time it is preempted.

$\mathbf{H}_i$	Set of threads with higher priority than thread τ_i.
$\mathbf{L}_k$	Set of threads with lower priority than thread τ_k.
$\mathbf{P}_{S_m}$	Set of threads which potentially may obtain the semaphore S_m during their execution.
$\mathbf{S}^n_{i,j}$	Set of threads which potentially suffer preemption by τ_k, instead of τ_i, and have not been covered at start of iteration n.
$\max(\ldots)$	The maximum value of the arguments.
$\min(\ldots)$	The minimum value of the arguments.
μ	Mean value of a random variable.
σ	Deviation of a random variable.
$\Theta^n_{i,j}$	The cumulative additional preemption delay caused by thread τ_j, on thread τ_i, after $n-1$ iterations.
$\mathcal{PAN}$	Path ANalysis tool: Developed during the preperation of this thesis, to analyse the control flow of a program, manipulating the object code to enforce the paths and control the measurements.

1 Introduction

1.1 Motivation

Modern general purpose processors are subject to various optimisation techniques aimed at improving average performance, and resulting in nondeterministic and/or, in poor worst case performance. Despite this tradeoff more and more of these processors are deployed in real-time systems. One reason for this trend is that the domain of real-time systems has grown considerably in the last few years, due to the increasing penetration of computer systems in every day life. Typical examples are cash dispensers, the microwave oven, or the up to 50 processors integrated in modern cars. Not all of these applications have real-time properties, and even less are life threatening, but if you take the example of a customer with his brand new car it would be completely unacceptable if his newly bought product ceases to work on the motorway, due to a deadline violation in the engine control unit. Due to this broadening application area the factors development, cost and time-to-market have gained considerably in importance. To use general purpose processors would be one way to achieve this.

Using measurements to estimate the worst case execution time is common practice in industry. However, up until now the common aproach for addressing this were ad-hoc and rule of thump methods. In most cases neither the test coverage, nor the real-time analysis are investigated. One of the main reasons for this is the tradeoff between the estimated probability of a failure on one hand and the cost and time consumed for testing and analysis on the other.

1.2 Contribution and Limitations

This dissertation contributes to research in this topic in several aspects. It shows how to provide reliable bounds on state of the art processors, not only on the WCET, but also on the delay induced by the preemption of task, utilising measurements. Path enforcement is deployed as a means to ensure that all path combinations possible in the later application of the software are surely covered in the testing phase. The complexity of this brute force method is reduced enourmously, by partitioning the program into several measurement blocks. The measurements are conducted utilising built in performance monitoring hardware of the processors. Such hardware is available on many high performance processors and is intended for performance optimisation.

To decouple consecutive measurements, and produce reliable results, the accceleration techniques of the processor are manipulated in such a way that either a worst case state, or a randomised state is provided at the start of each measurement block. In cases where neither is possible an analysis of the processor features delivers penalties to be considered to cover the potential negative effects. As with all real-time system, the programmer of real-time software is required to follow coding restrictions and guidelines. The restrictions imposed are necessary to avoid underestimation of the WCET. A typical example of such a restriction is the proscription of random access to memory through pointers, since the approach is not able to guarantee the WCET for such cases. The coding guidelines provide the programmer with hints to reduce overhead on the WCET. In some cases the gain in accuracy has to be bought with actual performance in some cases the coding guidelines even accelerate the code.

The approach relies in the current form on user annotations to provide information on the paths to be executed. However, techniques to automatically rule out infeasible paths and bound loop iteration can be easily integrated.

In order to avoid excessive overestimation of the worst case execution time, extreme value statistics are deployed. The previous work in this area, regards only the execution time of a monolithic program i. e.; one extreme value distribution describes the execution time of the program. Due to the partitioning of the measurement problem, the calculation of the overall WCET for a program is investigated. Since there is no analytical solution known to this problem, a empirical study is presented in this dissertation. A real-time analysis has been developed, that considers the complexity of state of the art processors. Over previous work this has the advantage of considereing the acceleration techniques of the processor in the preemption of a task.

1.3 Organisation of the Dissertation

The following Chapter will provide an overview on related work in the area of real-time systems while Chapter 3 comprises an introduction to real-time systems and possible monitoring techniques. The models of the embedding system, processor and software is the main theme in Chapter 4. Due to its high importance factor, the operating system software is more closely inspected. Furthermore, a statistical analysis of the worst case execution time, utilising extreme value statistics, is provided. Chapter 4 concludes with the scheduling proof for systems using state of the art processors. Finally, Chapter 5 is dedicated to the experimental validation of the previously described methods and also concludes this work.

2 Related Work

The task of estimating the worst case execution time of a piece of software can be divided into analysing the control flow of the software, estimating the execution time of a given sequence of instructions, and calculating the WCET based on the results of the two previous steps. Additionally, the analysis of the operating system itself, and the effects on the execution of the application software has to be investigated in context of a real-time operating system.

2.1 Control Flow Analysis

The potential control flow of a piece of software can be described using a *control flow graph (CFG)*. The CFG is usually extracted directly from the source, assembly or object code. The missing information in this control flow graph is the specification of feasible paths. A basic necessity is the bound on loop iterations. Additionally, infeasible paths may be excluded from the later search of the worst case execution time. Two basic principles are used in providing this information. Either the user is needed to specify the necessary information or symbolic execution of the code is deployed.

2.1.1 Annotations

Various papers have been published concerning the user supplied information on loop bounds and infeasible paths. One fundamental work referred to by many later publications is [71] by Puschner and Koza. In this work no actual annotations are used, instead the syntax of C has been extended to provide the information. A modified `for` statement is introduced which accepts the usual arguments of a `for` statement, plus additional information regarding maximum number of loop iterations or maximum amount of time for the loop, plus additional exception code to be executed when the given bound on time or iterations has been exceeded. Furthermore, *marker* and *scopes* are defined. A scope specifies a range of instructions in the code while a marker, within a scope, denotes how often this particular path may be executed through the marker within one execution of the associated scope. Markers are not allowed within nested loops. Using this system, the number of paths to be investigated are reduced considerably. A major drawback of the system is the necessity for many coding restrictions, and a simple underlying hardware i. e.; the processor utilises no caches and pipelines. The tool of Park and Shaw presented in [69] prompts the user for information on loop bounds, wherever needed.

Stappert, Ermedahl and Engblohm describe in [77] a more elaborate system where scopes and flow facts are used. The scopes provide a hierarchic description form of the program. The flow facts create restrictions to the possible path combinations like, for example, the number of times a given path may be executed within a loop, or the infeasibility of particular path combinations.

A symbolic expression of the number of loop iterations is used by Colin and Puaut in [20]. This allows the programmer to provide a closer bound on the number of loop iterations, for example, with non rectangular nested loops. In non rectangular nested loops the inner loops are not iterated to their maximum all the time. A classical algorithm providing this property is the bubble sort algorithm, where the number of inner loop iterations decreases with the number of outer loop iterations. The gain can be considerable depending on the nature of the program, but the additional complexity leads to an error prone process of annotating the program which is not easily verified since no functional property of the program is affected. Only a tedious manual check of the provided annotations, or a measured sample execution which exceeds the provided WCET, may reveal errors.

2.1.2 Symbolic execution

The deployment of symbolic execution as compared to manual annotations has the major advantage of avoiding the error prone annotation process, while the largest obstacle is the complexity of the problem itself. Gustafsson and Ermedahl describe in [22] a symbolic execution approach for a subset of C. In order to cope with the complexity, while at the same time avoiding serious overestimations, a mixture of value ranges and value sets are used. Whenever a value cannot be determined to a single value during data flow analysis ranges, or set of values is used as appropriate. A variable may also have both a set of values and a range of possible values.

In [58] and [59] by Lundqvist and Stenstroem the paths are simulated, but the values are either defined exactly or are marked *unknown*. No sets of values or ranges are used, thus the complexity is reduced considerably, even though this reduction is known to be of lesser accuracy. Whenever an `if` statement is encountered with an *unknown* in the decision statement, both paths have to be analysed. When two branches are joined, a conservative merge of the variable values is done. The memory and register state are joined, but only identical variables render a specific value for future reference of this variable, all others being assumed as *unknown*. Pipeline effects are considered at these joining points, by adding machine state information. As regards the caches at a join operation, the penalty potentially incurred by the cache state of the path with the shorter WCET is estimated. When this penalty, plus the WCET of the shorter path, is smaller than the WCET of the longer previous path the cache state is taken from the longer path. Otherwise only memory references contained in both joining cache states are considered. This results in many *unknown* references in the caches. At each branch instruction the path with the least progress is chosen for further analysis, thus ensuring that loop bodies are completed before iterating. The extensions to ranges or sets of valuables is relatively easy to implement.

Liu and Gomez use a similar approach in [57]. Here, as well, the value *unknown* is introduced to handle complexity and undetermined path execution. In the first instance, the program is transformed to carry additional parameters which describe the execution properties. These properties provide the number of primitives used by the program. The execution time of these primitives is measured once and this information is used, together with the symbolic evaluation, to provide the WCET. This approach is, however, unsuitable for processors with acceleration techniques.

A slightly different method is presented by Ernst et al. in [88], [87] and [91]. Symbolic execution is used in these papers to detect input dependent, and input independent parts of the program. These properties are used later on in hardware modelling, where input independent parts can be bound much closer to the real WCET than parts which lack this property. The parts are separately analysed by a cycle true processor simulator, when available, or with instruction timing addition for a simple processor architecture.

Healy et al. in [34] focused in bounding the number of loop iterations. First of all, those branch instructions that can affect the number of loop iterations are identified by utilising compiler techniques like computing the dominator tree. Then the conditions under which these branch instructions change direction are computed by utilising compiler optimization techniques like loop unrolling. Next follows a reachability analysis to determine in which iteration a certain branch instruction is actually met. For loop exit conditions, dependent on input variables and loop invariant, the user is prompted to provide bounds on the values of these input variables and the number of iterations is computed. Finally, nested loops, where the number of inner loop iterations directly depends on the outer loop iteration, are considered. The loop bound is provided using a symbolic summation expression.

Blieberger, Fahringer and Scholz in [11] use symbolic evaluation to determine the cache usage of an application. To demonstrate this, instrumented C code is evaluated symbolically, yielding a symbolic tracefile which indicates the memory references accessed. By symbolic evaluation a cache hit function, dependent on the input data, is computed.

2.2 Execution Time Estimation

In order to estimate the best and worst case execution time, we can utilize modelling and measurement based approaches. Since these approaches often focus on small portions of code, a third major thrust in research dealing with the computation of the execution time bounds of the whole program out of the execution time bounds of the parts.

2.2.1 Hardware Modelling

The work in this field can be divided into two approaches, that of modelling either simple processors or partial features of more advanced processor.

An analytical model exclusively for caches is the sole focus of Harper, Kerbyson and Nudd in [33]. This approach is limited to rectangular nested loops. In the first step, the cache footprint i. e.; the used cachelines of each memory reference, is computed. The memory references with identical access patterns are grouped into translation groups. Secondly, the cache footprint of the translation groups are computed. Finally, the interference is computed. The authors distinguish between spatial self interference, spatial cross interference and external interference. Self interference is caused when the members of a single array within one loop iteration are referenced in such a way that they preempt each other. The act of members of a single translation group preempting each other is called cross interference. External interference, on the other hand, is present when members of different translation groups force each other out of the cache. The restriction to rectangular loops considerably narrows the application field of this approach.

Narasimhan and Nilsen analyse a pipelined RISC processor in [67]. The presented pipeline simulator is fairly straightforward. The main focus of this paper is the aspect of portability with regard to which data must be provided in order to configure the presented analyser for a given processor.

The branch prediction unit of the Intel Pentium processor was investigated by Colin and Puaut in [20]. The caches provided by the processor are switched off. A classification of the control transfer instructions is introduced and fed into a branch predictor model. No actual modelling of the pipelines, or the execution time, was conducted.

Stappert and Altenbernd model a processor including caches and pipeline, but simplified the analysis by assuming straight-line code in [75] and [76]. According to the authors straight-line code i. e.; code without loops or recursions, is found quite frequently in code produced by automatic code generators. The analysis begins on basic block level and is then extended to handle inter basic block interferences. Part of this work is used as *local low level analysis* in [77] by Stappert, Ermedahl and Engblohm. Here, a 32 bit RISC micro–controller with pipeline, but without caches, is investigated. Loops are broken up and the basic block responsible for iteration or continuous operation is duplicated. Thus the pipeline analysis of the loop iteration is decoupled from the analysis of the loop exit.

Li, Malik and Wolfe in [54] model a processor with direct mapped instruction cache. The analysis begins at basic block level. Nevertheless, the execution time of a sequence of instructions is assumed to be time invariant, and subroutines are handled as if inlined for multiple calls to this routine. Thus the gain of the instruction cache by multiple calls of a subroutine is voided. A cache conflict graph is used to determine the number of cache hits and misses.

Coloured petri nets are used by Burns Koelmans and Yakolev in [29] to model a super–scalar processor. The instructions are modelled as tokens passing through the petri net. These tokens have source and target dependencies to model the register and memory reference usage. A small non-determinism example extends the model into 4 different execution states which shows the major weakness of the approach due to the fact that the state space explodes with only a small number of indeterminisms.

A case study for WCET analysis for a MIPS R3000/3010 is presented by Hur et al. in [36]. The processor has a pipeline and direct mapped data and instruction caches. Start of the analysis is done on basic block level. With each basic block the pipeline state at the beginning and end of the basic block, and the cache usage, is computed. The cache usage is provided as *first_reference* and *last_reference*. The term *first_reference* indicates the memory references that lead to a cache hit, if already in the cache, while *last_reference* provides the cache state after the execution of the basic block. When the paths are resolved the *last_reference* of all "executed" basic blocks are merged and used as input for the *first_reference* of the succeeding block. A very similar approach to these reference sets of basic blocks is used by Wolf and Ernst in [86]. The memory address of a data access is of the form baseregister plus displacement. The authors classify the data references into three types:

1. Global data which is accessed through the global base register i. e.; the memory location is known and fixed.
2. Local data referenced through the stack pointer where the actual memory reference is dependent on the calling structure.
3. Finally, all other accesses which are assumed to induce two cache misses. One cache miss due to the potential necessary load of the referenced data, and another for a useful cacheline preempted by this reference.

This has been extended in [50] and [52] to determine the intertask preemption delay in multitasking systems (cf. Section 2.4).

In [46] and [45] Kim, Ha and Min try to determine the overestimation sources of WCET analysis. They consider data caching, instruction caching, pipelining, effects across basic blocks (combines virtually the first three) and the effects of infeasible paths. To analyse the amount of overestimation introduced, by not considering the effects described above, they introduced some switches into the previously described MIPS R3000 simulator in order to be able to switch the consideration of these effects off and on. The impact of the infeasible path analysis is orthogonal to the rest and is therefore handled separately.

In [60] Lundqvist and Stenstroem address timing anomalies in modern processors. They show how a cache miss can speed up execution on processors equipped with out-of-order execution units. In their investigation, they used a reduced PowerPC Simulator derived from *psim* from Cygnus.

The work of Müller et al. starting with the dissertation of Müller [64] over [7] up to the recent publication [65] focusses on the simulation of caches with special regard on instruction caches. All cache accesses are classified into:

Always Miss : this memory access will never hit the cache.

Always Hit : this memory access will always hit the cache.

First Miss : within execution of this program/loop, the first access will miss and all subsequent accesses will hit the cache.

First Hit : within execution of this program/loop only the first access will hit and all other accesses will miss the cache.

For this classification *abstract cache states (ACS)* are introduced. These ACS describe all the memory references that might be in the cache at a given point of the program. It contains the addresses of the memory references and the corresponding age. The age is necessary for the least recently used preemption computation for the cachelines. Each basic block takes an ACS as an input and transforms it in accordance with the memory references inside the basic block. At a junction point, where two or more paths are joined, the ACS of all paths are combined in such a way that all memory references that are potentially held in a cacheline at this point are included into the the joint ACS. The modelled cache has only a size of 256 bytes.

The fundamental theory of the work above has been used by Theiling, Ferdinand and Wilhelm as a basis in [79]. They have developed three analysis steps to determine the type of memory reference with regards to the classification of memory references above. The *must analysis* abstracts the age of a memory reference i. e. the minimum lifetime of this reference. The join operation after alternative paths takes only those memory references into account, which are contained in the abstract cache states of all paths joined at this point. The age of these references is assigned from the age of the oldest i. e.; most likely to be replaced reference in the joint ACS. This allows the detection of always hits. The *may analysis* focus on identifying always misses. The join operator takes all memory references potentially held in the cache. If the memory references of two paths to be joint have a different age, the minimum age is taken. The *persitance analysis* is utilised to detect first misses. The persitance analysis is a blend between must and may analysis. The join operation takes all memory references as in the may analysis, but picks the oldest age of duplicate entries as with the the must analysis. Additionally, the persistance analysis tags memory references which have been removed from the cache. If a memory reference is not an *always hit*, but has not been removed out of the cache then it has to be a *first miss access*.

A Motorola PowerPC with pipelines and caches but without caches is the target platform for Hergenhan and Rosenstiel in [35]. Their analysing tool GROMIT takes assembler text, a simplified processor description and a path relation file as input. The latter provides essentially all necessary control flow information. A cache conflict graph and cache state transition graph are deployed for analysis of the cache behaviour. The authors provide a method to make a tradeoff between accuracy and simulation time. The output is an integer linear programming problem description (cf. Section 2.2.3).

The problem of insufficient and flawed processor documentation is tackled by Atanassov, Kirner and Puschner in [8]. To provide a reliable hardware model they utilise measurements to complete and either correct, or proof the information regarding the execution time of instructions as gathered out of the processor manual. The Infineon C167 used processor is relatively simple since it has no caches or pipelines. After building a model of the execution times of instructions this model is again validated against real hardware by executing programs and comparing the results with that of the theoretical work.

2.2.2 Measurement Approach

Measurement based approaches have experienced a revival within the last couple of years. Years ago the research focused on getting reliable information about worst case execution time by means of measurements. As processors tended to be more complex, using pipelines and having data dependent execution times for instructions, this was no longer regarded feasible, with the result that the theoretic models were developed. Now again complexity issues make those exact models almost infeasible for state of the art processors. Additionally, it has been perceived that the documentation of the processors is often inaccurate or incomplete.

A very simplistic processor without pipeline and caches is the target platform of Lindgren in [56]. The paths of the program are instrumented by `ADD` opcodes to specific variables identified with a path. Only a number of measurements with random input data are taken and, together with the path counter, provide an linear equations system which is then solved. By intelligent placement of these instrumentations the amount of variables can be reduced drastically. For example, a piece of linear code succeeded by an alternative needs only one counter for the linear code, and another within one of the two alternative paths. More elaborate variants are possible. The `gcov` utility of the `GNU` `gcc` compiler uses a similar technique.

The performance monitoring hardware provided by the Motorola PowerPC 604e processor is utilised to estimate the execution time in [61] by Corti, Brega, and Gross. The ultimate target of this approach is not to put an upper bound on the WCET but to approximate the execution time for a given percentage of executions of the program i. e.; a quality of service. The scheduler of their system will take care that a task using more than it's alloted time will be prematurely terminated. All instructions are measured under best case conditions. The results are used to provide a cycle per instruction metric. A given program is then run and the performance monitoring hardware counts the number of issued instructions, cache misses and mispredicted branches. This data is used to compute an estimate on the worst case execution time.

An extreme value statistics approach is described by Burns and Edgar in [13], [14] and [15]. Extreme value statistics are used in civil engineering in predecting worst case weather or flood scenarios. This approach takes no information about the nature of the executed program. A large number of measurements are taken and then an extreme value statistics probability density function is matched to the measured execution time. With this probability density function it is possible to provide an bound on the WCET according to a required confidence in this value. This approach relies solely on the statistical properties of the program since no worst case state and no worst case input data is used. Additionally, code and data is preloaded into the caches of the processor. The concept of extreme value statistics is explained in greater detail in Section 4.6.

Müller and Wegener compare an accurate timing model, as presented in the previous section, with a genetic algorithm testing approach in [66]. The underlying idea of the approach is to produce an estimate on the worst case execution time by trying to find the input producing the WCET. One necessity for a useful deployment of genetic algorithms on a particular problem, is the tuning of a large number of parameters. Here it is the number of input parameters to the program. A SPARC IPX architecture has been used for the experiments. The caches have been switched off, due to the fact that Rational *Quantify*, the tool used for the measurements, does not take the effect of caches into account. Quantify performs cycle-level timing through object code instrumentation. Thus overhead by the operating system was ruled out and repeated runs, with the same parameter set, resulted in identical execution time measured. The results showed that evolutional testings are unsuitable for providing secure bounds on the execution time of a program. This is due to the fact that the parameter/fitness space in the case of a program is a set of plains which may have no direct connection to each other, for example a given `if` condition, where an input parameter is compared against a fixed value, and is evaluated as either true or false. On the contrary, genetic algorithms work best with a contignous fitness/paramter space which may have a limited number of salti.

Writes to an address monitored by a logic analyser are used by Wolf, Kruse, and Ernst in [89] to determine the execution time. The write addresses named trigger points inserted in the source code have two functions. On one hand the trigger points allow for extraction of the path which has been taken by the program, on the other hand the logic analyser is capable of providing cycle true values for the execution time between two consecutive measurement points. Thus a segment wise measurement with exact control of the taken paths is possible.

2.2.3 Calculation Methods

Most of the approaches for WCET estimation are based on providing meta results as regards the WCET estimation of partial constructs of the program. These constructs may be individual opcodes, basic blocks or a group of basic blocks. The WCET of the program is computed using either tree based, path based or implicit path enumeration based approaches. Syntactical parse trees, as used by compilers, are usually used for the tree based approaches. Using a bottom up method each edge in the tree is assigned a execution time, and the overall worst case execution time is computed by summing up the edges according to the rules given by the control flow analysis. The work of Puschner and Koza in [71] resorts to this comparably simple method.

The path based approaches search all potential execution paths explicitely given for the one with the longest WCET. This method works well for flow restrictions within one nesting level, but gets rather complicated when the restrictions stretch across different nesting levels. Stappert and Altenbernd use this path based approach in [76]. As described in the previous section, the approach computes the individual WCET of all paths in non-looping software. After this, k paths which have the greatest WCET are checked for feasibility. If one or more feasible paths are found, the largest WCET is provided as WCET of the programm. Otherwise the shortest WCET of the k paths is taken as an upper bound on the WCET. This has been extended by Stappert et al. in [77] to handle each nesting level of a loop in itself. The flow information is used to limit the number of paths to be analysed. The Dijkstra algorithm is deployed to search for the longest path in the program.

Lundqvist and Stenstroem use a comparably simple technique for their machine modelling approach in [58] and [59]. To bound the number of paths to be investigated the largest WCET of all the paths joining in one point inside the program is taken. The handling of caches has been explained previously in the hardware modelling section.

As regards the IPET approaches, each unit, usually a basic block, is provided with a time variable $WCET_i$ and a counter x_i. With respect to the constraints reflecting the results of potential control flow, the following equation is maximized:

$$WCET_{sum} = \sum WCET_i * x_i \tag{2.1}$$

Li, Malik and Wolfe [54] were the first to name this technique. An example of the estimation of execution counts is that, the execution count of two sequential basic blocks are identical. Given an alternative, the sum of the execution counts of the alternatives equals the execution count of the preceding block. Further restrictions are loop bounds, or mutual exclusive parts. The result of this description is a large number of linear equations and linear inequalities. These can be reduced to an *integer linear programming (ILP)* problem, which may be solved using existing ILP solvers. Another example of an IPET based approach is the graph theoretical method of Puschner and v. Schedl in [72]. The authors assume the execution time of basic blocks to be known and constant. This excludes all processors with pipelines and caches. The graph is used for loop detection and finally mapped to an ILP problem.

Theiling et al. [79] use a virtual loop unrolling and virtual functions inlining method to enhance the accuracy of their execution time estimate in the presence of caches and pipelines. The result is still an ILP as in [54]. The approach of Hergenhan and Rosenstiel in [35] is similar.

The basic idea of the approach presented by Wall et al. in [82] is to assign each Ada-Source construct a worst and best case execution time, and to compute the WCET and BCET of the whole program by linear superposition of the WCET and BCET of the constructs. To identify the WCET and BCET measurements of reference programs with a consecutive regression analysis is used. As processor, an Motorola 68030 derivate was used which has not the problem of caches and complex pipeline structures.

2.3 Operating System Analysis

A general purpose operating system is the target of the profiling tool Kitrace presented Kuenning in [49]. By insertion of TRAP instructions in the kernel code, a provided routine is called whenever one of the instrumented pieces of code is reached. In the provided interrupt service routine, hardware counter are utilised to measure the time. In contrast to the approach presented in this work no steps are taken to produce the worst case situation instead the measurement objective is to produce average performance measures for operating system services. This approach utilised Motorola, Intel i386 and 68000 on Sun OS 4.1.1 as target processors.

The focus of Colin and Puaut in [21] lies on the applicability analysis of their WCET analysis tool on the operating system kernel. The main problems the authors encountered are the dynamic features in general, and of these especially the memory management.

The memory management allows dynamic memory allocation and deallocation. As regards the allocation of heap memory, the search tries to provide a continuous piece of free memory which suits the allocation request. This results in very pessimistic assumptions for memory allocation requests. A good part of RTEMS could not be automatically analysed by their tool, and had to be thoroughly manually analysed and then modified to be suitable for the tool. The overall result of the work is that automatic analysis methods can also be applied to operating system code, but that usually a lot of manual work is still necessary, since the constructs of the operating system supporting the dynamic features of the operating system are not easily accessible to automatic analysis.

2.4 Scheduling Analysis

In the literature a huge amount of scheduling schemes and corresponding proof methods are proposed. Most of these suffer from lack of support for non zero task switching times. These non zero task switching times have two origins. One is the time needed for the scheduling analysis itself, the other stems from the disruption of the working set within the caches and the stalls in the pipeline (for further examples refer Section 4.2.1) on modern processors. The consideration of schedulability analysis methods will be limited to those approaches supporting these effects.

Busquets et al. provide a simple formula, and the corresponding proof, for a simple fixed priority scheduling scheme in [16], [17], and [18]. A basic requirement for the applicability of this approach is that the deadline of a task τ_j is shorter than the interarrival time of the task T_j. The approach tries to allocate a time window w_i for the execution of the whole task τ_i, including the WCET of the isolated task C_i, all blocking times B_i, preemptions by tasks of higher priorities $\mathbf{H}_i$ and the corresponding penalty for the scheduling algorithm and disruption of the cache γ_j for each preemption. The cache preemption in this γ_j is computed solely by the cache usage of the preempting task. The preempted task is not considered for this computation. The resulting equation for the schedulability analysis is provided as follows:

$$w_i^{n+1} = C_i + B_i + \sum_{\tau_j \in \mathbf{H}_i} \left\lceil \frac{w_i^n}{T_j} \right\rceil * \left(C_j + \gamma_j\right) \tag{2.2}$$

An initial w_i^0 can be arbitrarily chosen. The formula computes a new execution window w_i^1 by considering all effects that may arise during the initial set time window. This process is iteratively repeated, until either the resulting window w_i^{n+1} exceeds the deadline D_i of the task τ_i i. e.; completion of task τ_i cannot be guaranteed before its deadline, or the window stops to increase i. e. $w_i^{n+1} = w_i^n$. In the latter case the execution window corresponds to the response time of the task i. e. $R_i = w_i^n$.

Lee et al. present in [50] and [52] an integral approach for modelling the processor with caches and the schedulability analysis. The cache related preemption delay is bound by inspecting exactly the cache behaviour of the different tasks, and compute the amount of cachelines of task τ_i which might actually be dislocated from the cache due to the preemption of task τ_i by τ_j. The interactions are presented as an equation system which are solved by employing an ILP approach.

3 Real-time and Monitoring

3.1 Real-time

The term *real-time system* is often confused with *extremely fast system*, but a real-time system is actually defined by time constraints which can be in any given order of magnitude. Most technical systems, have such time constraints which usually take the following form:

The result has to be delivered within x seconds after the triggering event!

Such time constraints are called *relative deadlines* or simply deadlines. Examples of such deadlines can be as short as some tens of nano seconds for the inter bit timing of a 1 MBit/s CAN controller (Controller Area Network) or as large as several hours for a weather forecast system.

Because of the central role of time constraints in real-time systems, these systems are classified according to the type of time constraints. The first class of real-time systems are the *soft real-time* systems. This class covers those systems where a miss of a deadline results in degraded performance, and/or the increasing of costs. Examples of this kind of deadline would be, for example, the response time of an entertainment system (e. g. car radio) after pushing a button, or a stall in a production line due to a communication timeout and retransmission. While the first leads to reduced customer satisfaction and a possible loss in market share, the second leads to direct loss of production output (return of interest). Most systems belong to the soft real-time systems. Figure 3.1 shows the development of costs versus time in real-time systems. With soft real-time systems (line 1 and 2), the miss of a deadline simply leads to an increase in cost. In the case of most of those systems, a deadline miss can be tolerated, as long as the probability of such a miss is less than a given value.

The second class are the *firm real-time* systems. In these systems, the miss of a deadline leads to the result becoming worthless. An example of such a system is a weather forecast system, where the result becomes void no later than when the forecasting horizon has passed.

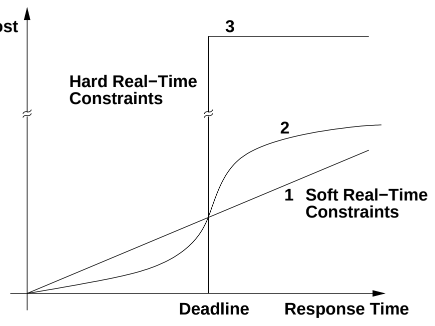


Figure 3.1: Cost Function of real-time-Tasks [30]

The *hard real-time* systems build the third and last category. A miss of a deadline here is followed by more or less catastrophic consequences. The consequence is usually loss of life or, at the very least a serious amount of money. We think here of an X-by-wire system in vehicles and planes, or of simple things like the destruction of a machine and the following halt of a complete production line. Line 3 in Figure 3.1 shows the enormous jump in costs. A deadline violation is considered fatal and cannot be tolerated.

3.2 Monitoring

A monitor is a tool, or a set of tools, that facilitate analytic measurements in observing a given system. The goal of these observations is usually the performance analysis and optimisation, or the surveillance of the system. The reason for the latter may be a measure of safety to avoid critical system states or unauthorised access. In the following the focus will be set on performance monitoring as part of the development process.

There are a number of major questions which have to be answered in order to to decide which monitoring scheme should be used.

1. What information is pertinent to the measurement objective?
2. When or under what condition should the information be gathered?
3. How to obtain the desired data and what is the impact of the method?

Identifying the measurement objective, and the relevant data needed to reach this objective is the first step to building a monitor. In the literature the focus is often set on measuring and optimising the performance of multiuser computer systems (e. g. [24], [78], and [25]). The relevant data in this case is the number of user processes running concurrently, the amount of computation time consumed by the operating system, and the amount of time spent waiting for external hardware (communication and IO).

Generally, the objective is to gather information about either system states or events. In the case of events, this may be the number of events in a given interval or the inter arrival time between events. Examples of such monitor events are the occurrence of an interrupt, or access to a particular piece of hardware. In the case of system states, the basic factors are the probability or the duration of a specific state. Typical examples of relevant system states may be the number of users in a multiuser system, or the execution of the idle task in a simple multitasking system. The work of Ferrari et al. in [25] focuses on state observation for tuning and optimisation purposes. Systems with a state based monitoring objective can be treated as an system with event objective, whereas a change in the system state is defined as an event.

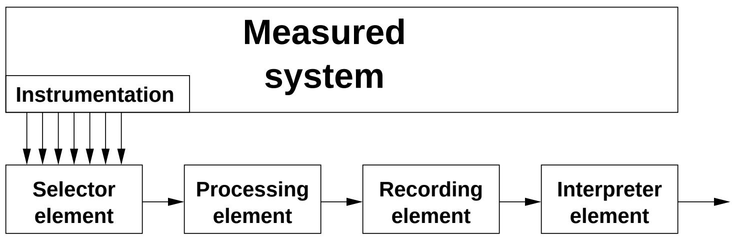


Figure 3.2: Structural Elements of a Monitor [78]

Figure 3.2 shows the typical structure of a monitoring system. The system under investigation is instrumented in a way, to produce the data needed. That may be for example some additional code in the software, or the connector to the system bus. The selector element serves as a filter, focussing the measurements towards specific relevant data. In the case of the bus-monitor, this may be the detection of access to a particular memory address. The processing element brings all the relevant data together and in some cases performs data preprocessing, while the recording element is used to store the data to a suitable device. This may be, for example, a memory portion or a hard disc drive. The data stored consists usually of the event or system state type and possibly a times tamp and/or additional values of variables, or registers of interest. The interpretation and visualisation of the accumulated measurement data is usually done off line. In cases where this is done online, it is often executed on an additional computer, to avoid the commonly large impact this part of the operation would have on the measured system.

3.2.1 Event Detection

The question “when to sample” is closely related to the potential data to be gained from monitoring. Detecting an event to sample the relevant data is one possibility to achieve this. Such a monitor is said to be *event-driven* and the data produced by this mechanism is called an *event trace*. Events can be classified into software and related events called *software events* and *hardware events*. A software event occurs when one of a set of certain points in code is executed, e. g. the start of an I/O-routine, or the value of a variable being within a given range. A hardware event may, or may not be, directly dependent on the logical content of a piece of software executed but consists of the appearance of a signal constellation in the circuitry of a system component. Typical examples for hardware events are a cycle stealing DMA request or the change of state of a photo sensor. Many hardware events generate a corresponding software event like an interrupt request and vice versa a specific piece of code which is recognisable as an opcode fetch of a specific address.

It is usually easier not to build an event detector for every single event type but to trigger instead on event groups. This may be the occurrence of, for example, any interrupt instead of two or more specific ones or simply avoiding to trigger on a complex signal pattern and using a less complex pattern while taking into account that there will be an additional registration of events which are of no interest regarding the measurement objective. This corresponds to a strongly reduced functionality of the selector unit in Figure 3.2 and is called *full trace monitoring*. The concept of full trace monitoring has several drawbacks. First of all the impact on the system under test has to be considered. In addition to this, the amount of data to be processed increases very quickly thus producing problems (e. g. the rapid filling of available buffer space and the problem of pushing the data to a mass storage system). In order to avoid these problems, it is usually reasonable to make the event trace selective. But even without full trace monitoring the amount of data accumulated is often considerable.

There are two basic fields of application for event driven monitors. The *count mode* and the *interval mode*. In the count mode the number of events of a given kind, which occurred in a given interval, are of basic interest. An example of such measurement objectives would be to get a statistical grip of interrupt activity. In the interval mode for each event the type of the event and the time, this particular event occurred, is stored. The time is usually relative to the first event or system start. A sample application field is used to extract the amount of time a system is persistent in a given state.

3.2.2 Statistical Sampling

The design of an event detection mechanism, as described in the previous section, is usually very time consuming. As an alternative method statistic sampling can be utilised. Unfortunately as the name implies only statistical statements can be derived from these, i. e.; it is only applicable when the exact sequence and number of occurrences of a given state is not needed. In contrast to the event monitoring, which provides all the relevant data for the monitoring objective, only a subset of the system is taken into account. This subset is defined by the points in time the samples are drawn.

The state of the system may be sampled either in periodic, or in statistic intervals, or in periodic intervals with a statistic deviation. While the periodic intervals have the problem of potential aliasing effects in the observation, the mere statistical approach suffers from the fact, that a local burst of samples may produce a completely wrong view of the systems properties. To avoid both problems the two methods are often combined using periodic intervals with statistic variation. Thus the aliasing effect is avoided while observing the system. The major advantages of this method is the usually simple implementation of such a monitor, and the comparably small impact on the system. On the other hand, the method degrades considerably in the face of system states, which are seldom reached or which are only observable for a very short time. In the presence of such states only two ways remain to observe and gain statistical relevance for these states. Either the sampling rate or the test interval have to be increased. While in the first case the impact on the measurement object will usually rise, the necessary extension of the test interval is often very painful and not feasible. In both cases the data volume that needs to be handled increases considerably.

Since statistical sampling can be done using an event driven monitor in combination with an induced event load, the main focus in the further discussion will be laid on event driven monitors.

3.2.3 Hardware Monitoring

The main characteristics of a hardware monitor is that they are external to the system under test. The connection of hardware monitors to the system is realised by high impedance *probes* (see also figure 3.3). Thus hardware monitors for digital systems do not need any of the resources of the system under test and usually imply virtually no influence to the measured system at all. However even the high impedance probes of the monitor add to the electrical load to the signal and, therefore alter the timely behaviour of the signals slightly. In modern, fast computer systems, this minimum change at critical points of observation can lead to random errors or completely inhibit the system's operation. These critical points obviously need to be eliminated from the list of observable signals. Thus, generally, it can be assumed that hardware monitors are systematically induced and more accurate than software or hybrid monitors.

The probes may be connected to *concentrators*, or in simpler monitors fed directly to the *logic modules*. The concentrators collect the various signals from the probes and transform these to signals compatible with the tools internal structure. It is often possible to switch the impedance of the probes as deemed necessary, by sending control commands to the concentrators. The logic modules implement the event filters by logically

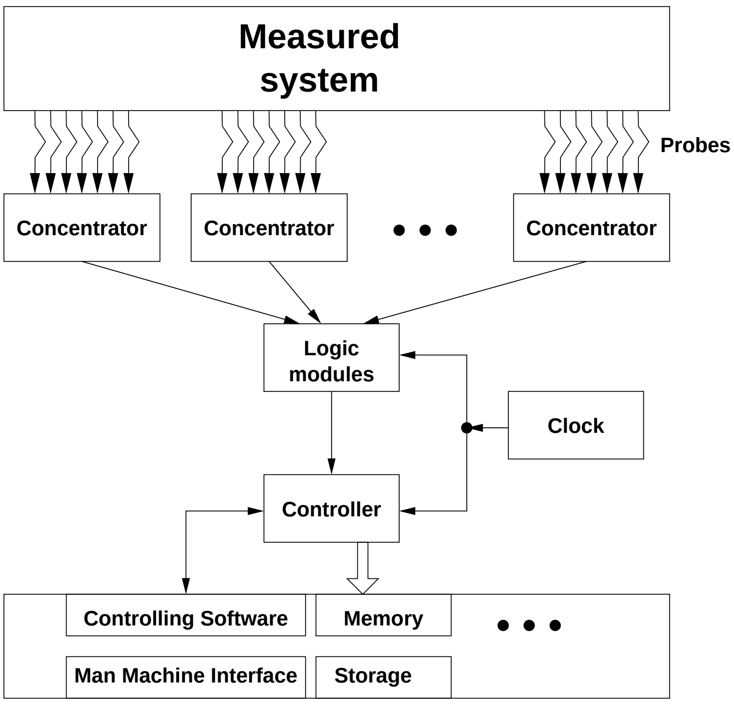


Figure 3.3: Elements of a Hardware Monitor [25]

combining the signals from the probes as specified by the user. The complexity of the capabilities of the logic modules varies vastly from detection of a simple combination of a number of input signals, the combination of input signals to numerical values and tracking of the, for example, *less*, *greater*, *between limits* type, to the congregation of rather complex signal patterns over time. In older systems the logic elements were basically plug boards, while in modern systems programmable logic devices like FPGAs programmed at reset by static RAM are used. The clock in Figure 3.3 is necessary for synchronising the incoming data. The task of the *controller* is to transfer the relevant data to the main memory.

Hardware monitors are often built on the basis of personal computers. This has several advantages. First of all, this frees the developer of a hardware monitor from having to design of the basic system which usually consists, besides other less important things, of a user interface, storage facilities and considerable memory space. The storage facilities are usually needed to file away the enormous amount of data for later processing which often exceeds an economical legitimate amount of RAM. To fill the gap between the main memory and the controller, the controller often implements a small buffer and transfers its contents via direct memory access (DMA) to the main memory.

Logic analysers are often used as implementations of hardware monitors, since they are equipped with all necessary elements i. e. the probes, triggering logic, memory and the user interface, but a given logic analyser is only suitable as a hardware monitor if it provides the essential elements of counters or interval measurement.

An example of a hardware monitor is described in [47]. Klar et al. focus on monitoring of distributed systems by utilising the self built, event driven hardware monitor *ZM4*. For evaluation and correlation of the measured data the tool *Simple* is deployed.

3.2.4 Software Monitoring

As the term software monitors implies these tools are based on software technology as opposed to hardware monitors. To make a software monitor work, extra code has to be inserted into the system under test. There are three possible ways to do this:

1. Modification of the software under test.
2. Modification of the operating system or system task.
3. Addition of a task.

The method chosen depends on the measurement objective. In the following an exclusion is assumed, whenever the method is not suitable for measurement. The third method which simply involves adding a task or program is usually preferred, since it is easy to add if required and easy to remove if not. In this way, the integrity of the application software and the operating system is maintained. This kind of tool requires appropriate interfaces to application software and operating system to gather the needed data. An unfortunate limitation of this method is that it will almost always be restricted to the class of monitoring objectives that allow sampling techniques.

The second method is used whenever the required data is not accessible to user tasks. Advantages of the extension portion in the operating system are the accessibility of the memory of all tasks and the easy transition from measurement to production code for the application software. A major reason for non-applicability of this method is that commercial real-time operating systems are usually delivered without source code and thus a developer has no means to implement the necessary extensions.

The last possibility, or first method, can also be seen as the use of *software probes*. Similar to the hardware monitors, the probes are placed at critical points in the software to be measured. The freedom of this process puts forth an enormous application area, however, the placement of these probes is critical, time consuming and must be redone with every new application under test. Another problem is that of accessibility of the necessary hardware. Modern processors often supply clock cycle counters, which are very useful in getting an exact time for software monitoring. However, these cycle counters are, in many cases, only accessible in the supervisor mode of the CPU. This mode is often reserved for the operating system. In the case of real-time systems the application software is often organised in threads rather than in tasks. The basic difference between a thread and a task is that each fully fledged task has its own separate and protected memory area whereas threads share their memory. The threads of a real-time systems usually run without memory protection, in the supervisor mode, to avoid the considerable task switching times implied with the memory protection and the change of the priority level while evoking system calls.

Commercial operating systems, even without working with the thread mechanism, allow the implementation of *interrupt service routines*. Thus the necessary access can be implemented via a trap mechanism i. e. a software generated interrupt. The use of a function call instead of a trap within a thread based operating system allows greater accuracy since the switching time from user to kernel mode is afflicted with indeterminism.

The ease of obtaining descriptive data is an outstanding advantage of software monitors. Because of this, much cryptic post processing needed to reconstruct the data can be avoided. Since the software tool can access the main memory, usually all relevant data can be gathered. As opposed to this hardware monitors depend on the information that passes on system buses.

Software monitors have, in contrast to hardware monitors, the undesirable effect of altering the system under test in a non-obvious way. This change in the system is called *monitoring artifact*. In general, the changes introduced by the monitor to the system require memory, possibly I/O resources, CPU time and alter the state of the acceleration techniques of the CPU (cf. Section 4.2.1).

Further information to software monitors can be found in Section 5.2. of [25].

3.2.5 Hybrid Monitoring

Hybrid monitors combine the advantages of both software and hardware monitors. Event detection is implemented using software probes. Thus, the great flexibility of software monitors is maintained. The preprocessing of relevant data is implemented in the hardware part of the monitor, and therefore reduces the impact of the monitor on the system under test.

The event detection is realised in two parts. One is by the insertion of additional code like in the software monitors described before. These probes trigger the hardware part either by a set of general purpose I/O pins of the processor, or by a memory access to a specific memory. The latter can be used to provide additional information to the hardware monitor, where a write access is usually used for that purpose. The hardware part combines this information with the data gathered in parallel by hardware probes, and is also responsible for the recording of the gathered data.

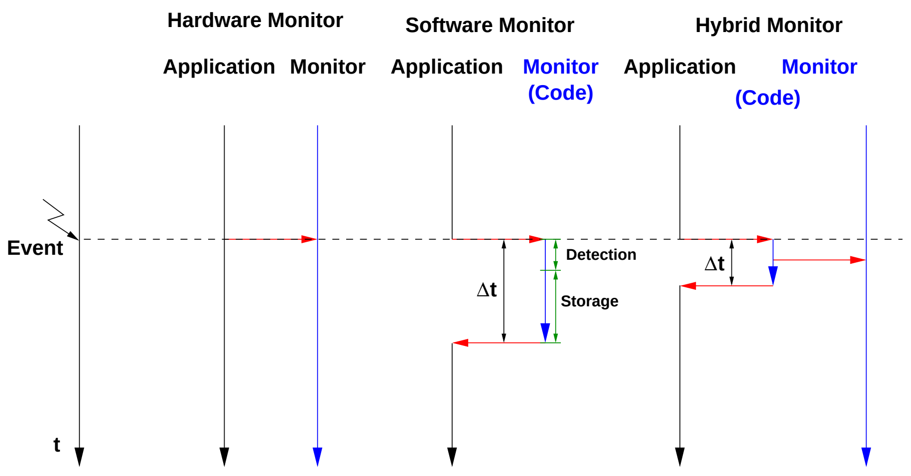


Figure 3.4: Temporal Impact of Monitor Types [63]

Figure 3.4 depicts the impact of the different monitoring schemes during the time the execution of the application is suspended. The hardware monitor induces no such impact, whereas hybrid monitors generate a rather small suspension time on the application software and software monitors suffer considerably from this effect. It should be noted that the actual suspension time is strongly correlated to the amount of data gathered in the software part of the monitor.

Liba Svobodova covers hybrid monitors in greater detail in Section 6.7. in [78].

Despite the greater accuracy of hardware monitors, the simpler application of software monitors and the blend of interests in the intermediate hybrid monitors a comparison can not be made assuming they had equivalent fields of application. Some monitor tasks can only be performed by using software and some only by utilising hardware monitors. Even if all monitor types were possible, one monitor type would usually be superior to the other from either a technical, or economical point of view. This is induced by the different types of events, which must be detected, and the varying depth of information needed.

The measurement objective is to retrieve the worst case time of a given piece of code. This excludes the sampling monitor from the list, since it is only suitable for retrieving statistical information on the run-time of code. Event driven monitoring on the other hand is able to provide the necessary information i. e.; the interval time between two measurement points within a program.

A basic desired functionality of the measurement is to provide the given piece of code with a worst case scenario of processor state. Taking this into consideration, one can distinguish between two kind of processors:

- Processors with caches and other acceleration techniques. In general, processors with acceleration techniques like branch prediction and speculative execution use these techniques only in addition to caches. These kind of processors are used whenever a serious amount of computational work needs to be done. This might be a driving assistance system or a telecommunication calling centre.
- Processors providing only simple pipelining, or no acceleration techniques at all. These are used often in mass market products with low computational demands like mobile phones (excluding the digital signal processing part) or the engine control for a car.

The final decision as to which kind of monitor to choose, strongly depends on which of the classes described above the processor belongs to. In the case of a simple pipelining processor like the Motorola MC68332, there is usually no need for complex actions to provide the worst case state of the processor. On the other hand, the execution of a instruction at a specific address in memory can be easily tracked by utilising a hardware monitor. To minimise the impact of measurement for this kind of processor hardware monitoring will usually be chosen. Logic analysers are utilised in most cases like this, in order to avoid having to develope a specific monitor.

To provide the worst case state for one of the more complex processors, it is usually necessary to insert additional code into the software under test. A hardware monitor might be able to detect the start and end of a piece of code on some processors, but the additional inserted code necessary negates the advantage of not having to instrument the software. The major advantage of a hybrid monitor is in reducing the impact of the software instrumentation. The advantage of this monitor for software detection and hardware sampling is again made void by the additional instrumentation used to provide the worst case state of the processor.

A sepcial feature available with many modern processors makes a different hybrid monitor possible: hardware detection and software sampling can be implemented by utilising debug registers, which generate an interrupt whenever an opcode at a specific address is executed. The following processor descriptions will therefore focus on relevant facts for event driven software and hybrid monitors.

4 Models

For an embedded system, there are three basic parts, which need to be modelled. One is the software which has to cover everything from application software to run-time system. Another is the hardware the software is running on i. e.; the controlling computer with processor busses and peripheral units. The last is the environment the system is embedded in. While the former mentioned models are essential for the measurement method itself, the last is only of marginal relevance.

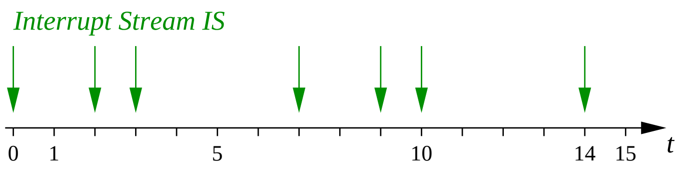
4.1 Model of Embedding Process

Real-time computing systems are usually deployed as the controlling instance of parts of their environment. One can think of examples like a drive by wire unit in a car, a quality control in an assembly line, or a video based obstacle avoidance system on a mobile robot. The connections to the environment consist of a number of sensors and actors.

In general, the deadlines are specified as the interval from detection on a set of inputs, to the reaction on a set of outputs. The processor is involved only from the time an input is registered, which is usually an interrupt, to the point where the write command to external hardware is issued. To perform a real-time analysis, shortened deadlines have to be used which only cover the time between the assertion on an interrupt and the write command. To determine the extreme response times computed by the real-time analysis (see Section 4.7), the maximum number of interrupts within a given interval is needed. This is not only limited to interrupts issued from external sensors, it must also be applied to interrupts of components of the computer system itself like, for example, timers, the hard disc controller or the keyboard.

One method of providing these values is by specifying the minimum *inter arrival time* between two interrupts of the same type. For the real-time analysis which is exemplified in Section 4.7 it is assumed that these interrupts appear at the rate specified by this minimum inter arrival time. Situations where interrupt bursts have long intervals between each burst lead to severe over-estimation of the induced load on the machine.

To circumvent this drawback the notion of *event streams (ES)* was introduced by Gresser [31] and [32] and further utilised in [70]. In order to avoid confusion with the term *event* of the monitor nomenclature and due to the fact that these kind of events usually generate interrupt requests, this descriptions will further on be referred as *interrupt streams or IS*. Interrupt streams are a formal means to express the maximum possible number i of interrupts of a certain type within an interval a_r, with the cycle time z_r.



$$IS: \left\{\binom{z_r}{a_r}\right\}_{r=0}^{3} = \left\{\binom{7}{0},\binom{7}{1},\binom{7}{3}\right\} \quad (4.1)$$

Figure 4.1: Interrupt Stream Example[70]

An example of a possible interrupt stream and its notation is given in Figure 4.1. This corresponds to the following statements:

1. At most 1 interrupt within interval 0 i. e.; no simultaneous interrupts, as there is only one tuple given with $a_r = 0$.
2. Within any interval of 1 time unit chosen of the stream there will be at most 2 interrupts. This is denoted by the number of tuples with $a_r \leq 1$.
3. At most 3 interrupts within any interval of 3 time units.
4. The pattern is repeated no sooner then every 7 time units.

The simple case of an interrupt with only a minimal distance can be expressed by a single tuple specifying the period and an interval of 0. A tuple with a period $z_r = \infty$ can used to model unique interrupts after power on or during a long period of inactivity. In [31] to take an example, an elevator is chosen where initially the request buttons at all levels may be pressed simultaneously. Afterwards the request buttons lead to no more interrupts until the previous request has been served.

In order to provide the maximum possible number of interrupts of type i within a given interval T the following equation is used:

$$E_i(T) = \sum_{r=1}^{s} \begin{cases} 0 & : \quad T < a_r \\ \left\lfloor \frac{T-a_r}{z_r} + 1 \right\rfloor & : \quad T \geq a_r \wedge z_r < \infty \\ 1 & : \quad T \geq a_r \wedge z_r = \infty \end{cases} \qquad (4.2)$$

An important restriction to the design of the interrupt stream tuple, is that they have to provide the maximum possible number of interrupts when fed to Equation 4.2. Thus the following set of tuples would be inadequate to describe the event stream depicted in Figure 4.1:

$$IS = \left\{ \begin{pmatrix} 7 \\ 0 \end{pmatrix}, \begin{pmatrix} 7 \\ 2 \end{pmatrix}, \begin{pmatrix} 7 \\ 3 \end{pmatrix} \right\} \tag{4.3}$$

To achieve this the distances in the tuples have to fulfil Equation 4.4.

$$\forall_{1<r<s} : a_r - a_{r-1} \leq a_{r+1} - a_r \tag{4.4}$$

$$\forall_{1\leq r\leq s} : a_r < z_r \tag{4.5}$$

where s is the number of tuples in event stream IS. For the real-time analysis all possible interrupt sources have to be described by such an IS.

4.2 Basic Hardware Model

Real-time system hardware can be divided into the processor and the peripheral hardware. The border between the processor and the peripheral model depends strongly on the used hardware. In the case of PC style hardware, a lot of peripheral components need to be placed in the processor model.

The models have to be constructed taking the following two aspects into account:

- To what grade can the worst case be established during measurement? How is this achieved?
- How can we consider in the computation of the WCET bound, those parts of the worst case state, which cannot be manipulated to provide the worst case?

The information gathered thereby must be used later during instrumentation and WCET computation. Sections 4.3.6.2 and 4.4.6.2 shows the analysis results for Intel Pentium III and AMD Athlon as regards to the worst case state.

As previously stated, the PC and therefore PC style processors are used more and more in computing intensive automation application fields despite the fact that PC style processors are not build for these kinds of operation and the usually requested reliability. There are three major reasons supporting this trend:

1. Computing power is comparably cheap with PCs.

2. The PC's flexibility allows for easy extensions in the future. This, combined with the previous point, can be used whenever a new application exceeds the computing power, or memory capabilities of the system, to replace the base system with a newer one without much effort.

3. Standard operating systems with real-time extensions can be deployed. This is especially useful since the necessary drivers for a wide variety of hardware components are available. In addition, the look and feel of the graphical user interface *(GUI)* is well known by the users from their home computers.

A further reason for investigating these processors is their complexity. A method that works for such an unsuited processor will likely be easy portable to processors of lesser complexity.

In the following the basic necessities for a processor and peripheral model will be shown. In Section 4.3 these model building processes will be discussed for the *Intel P6* architecture. The next Section will then display the similarities and differences of the Intel P6 and the AMD Athlon family. A more general and more detailed analysis of the Intel P6 core and the AMD Athlon can be found in Appendix A.1 and A.2 respectively.

4.2.1 Model of Processor

The processor is a major theme in the modelling of the system. The first step, is has to define the boundary between processor and peripheral hardware. The level of detail of the processor model is variable. However, as a basic line of thought one can assume that the more accurate the model, the more reliable the results will be, with less overestimation being needed. It will be essential to know at least the basic structure of the processor. A minimum requirement would be to know the processor an assembler programmer level, i. e.; knowledge of addressing modes, instruction and register set. The first questions that have to be asked are summarised as follows:

- Is the execution time of a single assembler instruction variable, or is it fixed?
- Can additional executed code accelerate the execution of the rest?

Whenever an execution of all single assembler instruction is fixed, the values can usually be found in the documentation. In this case, the measurement could be replaced by simply formulating how often a individual instruction is executed and fed into a linear equation system (e. g. [71], [54] or [91]).

There are basically two reasons why processors demonstrate the behaviour of variable execution time:

1. The execution time is data dependent. This is often the case for multiply and division operations.
2. The execution time is dependent on data availability, context and/or due to statistic deviations.

While the first reason is usually solvable by adding a penalty to the execution time, the second source of execution time deviation has to be considered thoroughly in order to provide save bounds on the execution time. Unfortunately sometimes both sources of a variable execution time apply. Context dependent execution times, i. e.; the execution time depends on preceding and/or succeeding instructions which are usually causally determined by instruction caches or even simple pipeline structures like, for example, in the Motorola CPU32 core. Data availability can be responsible for deviations due to the effects of caches for the data or the different latencies of memory areas, for example, memory mapped I/O, flash memory or static RAM. Statistic deviations are possibly induced by asynchronous communication, for example, outstanding transactions on Intel's GTL+ bus. These execution time deviations are sometimes aggravated and sometimes attenuated by parallel and/or out-of-order execution.

Modern hardware uses a variety of acceleration techniques to bridge the performance gap between main memory and the processors. The throughput has evolved differently for memory and processor. Whereas 20 years ago the memory available usually had no problems keeping up with, for example the 4.77 MHz Intel 8088 processor, released by IBM on August the 12th 1981 (IBM 5150 PC Personal Computer), today the core frequencies have been tuned to about 300 times the 4.77 MHz, and the memory has only accelerated at a factor of about 50 (assuming modern DDR SDRAM).

To keep the execution units fed with instructions and data, techniques like out-of-order execution, branch prediction and last but not least caches have been introduced. All these techniques lead to the fact that additional code can either slow down or accelerate the execution of a piece of software. The following gives a short list of some of the possible execution accelerator techniques, and the relevant information regarding the measurement approach.

Memory Address Computation To support software independency of the number of processes running concurrently and the amount of memory available on a particular computer, the addresses used by the software are virtual addresses which are mapped at run-time to the physical addresses available on the given hardware. This mapping consists mostly of a number of look-up tables and a few computations.

Caches Caches need to be described in terms:

- type of cache: instruction, data or unified.

- size.
- organisation i. e. associativity, cacheline size, replacement scheme.
- refill time regarding a possible write back.

A good description on general cache design with respect to the PC cache architecture is given by Mazzucco in [62].

Branch Prediction The branch prediction scheme has to be known to such an extent as to be able not only to identify the worst case state, but to provide this or at least add sufficient penalties.

Pipeline The effect of instrumentation to the pipeline has to be taken into account. This could be of relevance if the conditional and the unconditional branches make different use of the pipeline. In general it must be ensured that all instructions are executed, in full, inside the measurement interval, i. e.; the first instruction just enters the pipeline at the start of the measurement, and the last instruction has just left the pipeline at the end of the measurement. In addition, the order in which the pipeline is fed may not be changed.

Unfortunately the documentation of high performance processors is often insufficient and incorrect. To provide save bounds for the WCET the model extracted out of the documentation must be verified and refined by various measurements on the real hardware. The test suite generated by these tests should be stored in order to verify the processors behaviour if the processor is replaced with a presumably similar one.

4.2.2 Peripheral Hardware

The components of the peripheral hardware vary vastly with processors and application fields. Some can be viewed in isolation, some have to be considered interactively. Peripheral components required by the processor should be modelled with it. Examples of such components could be the GTL+ bus of the Intel P6 family connecting the processor to the *host bridge* also known as *north bridge*. The host bridge is the component connecting processor via GTL+ bus with the main memory, the *advanced graphic port (AGP)*, the PCI bus and the *south bridge* providing connectivity to e. g. ISA and IDE bus, parallel and serial ports, to name but a few. While the GTL+ is a necessary component, the host bridge belongs to the *chip-set*. For the Intel P6 family more than 20 different chip-sets are available. The following gives a few examples of peripheral components and the detail of knowledge necessary. To actually provide sufficient models for all PC components would exceed the scope of this work.

4.2.2.1 Main Memory

The main memory is an essential component of all computer systems. In some controllers the main memory is integrated on the controller chip, but more performant systems have external main memory which is usually *random access memory (RAM)*. The memory type ranges from *static RAM (SRAM)*, *dynamic RAM (DRAM)* and *synchronous DRAM (SDRAM)*, through to *double data rate RAM (DDR SDRAM)* [44]. Due to the current major relevance of SDRAM the focus is set on these. In SDRAM the memory is comprised of capacitors which have to be recharged after a read operation in order to retain the memory state. The capacitors used for data cells tend to bleed off their charge, and therefore require a periodic refresh cycle or data will be lost. A refresh controller determines the time between refresh cycles and a refresh counter ensures that the entire array is refreshed. Of course, this means that some cycles are used for refresh operations, and therefore have some impact on performance. The refresh cycle is hidden from the processor due to the fact that access of memory works in a decoupled request/serve manner, i. e.; a data request is issued and the processor is able to do different things until the data is provided by the memory. Typical values for such a refresh stall is 1 cycle of the main–board frequency – which is often a fraction of the processor core frequency – every 10 milliseconds.

4.2.2.2 Timer

Time is an essential value in computer systems. While high performance processors provide cycle counters, usually timers are used to keep track of "external" time. In the case of PCs the *real time clock* integrated circuit 8254 (cf. [38]) is used. It consists of 3 separate 16 bit counter operating with 10 MHz. The counter can be programmed with 6 different counting modes. Of major relevance is mode 1 which is a hardware retriggerable oneshot and issues a timer interrupt at a given time whereas mode 3 generates timer interrupts with a given period. For our case the granularity of 10 MHz and the time needed to program the timer is essential. Since the timer is accessed by a system bus and special port I/O commands, the programming time has to be measured with the given hardware. For exemplary results see Section 5.3.1.

4.2.2.3 Interrupt Controller

The number of I/O pins is always an issue in processor development. Thus the number of interrupt pins is often comparably low. On the other hand external hardware is often not equipped to communicate with any given processor in order to issue an interrupt. To get "simple general purpose" hardware connected to an interrupt on the processor usually external interrupt controllers are used. In the PC world two cascaded *programmable* *interrupt controller (PIC)* 8259A (cf. [37]) are commonly used. An 8259A provides 8 interrupt pins. Each pin can be assigned an arbitrary interrupt priority level which is used for internal arbitration and is transmitted to the processor in the case of an interrupt. Whenever the PIC has issued an interrupt to the processor it waits for an acknowledgement before issuing another. With this approach the latency time between receiving and issuing an interrupt, and the time needed for the acknowledgement, is of relevance (cf. Section 5.3.3).

4.3 Intel Pentium III

The Intel Pentium III is one of the newer members of the Intel P6 family. The following will give a very brief summary of relevant facts of the Intel Pentium III. A more detailed description of the entire Intel P6 family is provided in Appendix A.1. Within this discussion the Intel P6 family is quoted instead of the Pentium III, whenever a fact is also true for the other members of the family.

4.3.1 General Architecture

One of the basic features of the Intel P6 family is its micro code core. The programming interface is CISC architecture while internally RISC code is executed. The CPU core of the Intel P6 family processors is divided in five parts. Figure 4.2 gives a coarse overview of the architecture. The *fetch/decode unit* translates up to 3 external CISC instructions into the internal RISC code within one CPU cycle. It is a *in-order* unit and contains also the branch prediction unit which is described in greater detail in Section A.1.4. The RISC code is called micro ops and contains two sources and one destination within one command word.

The micro-ops are stored in the *instruction pool*. The instruction pool can hold up to 40 micro ops ready for execution, or already executed but not yet committed to machine state. The effective execution of the micro ops is done in the *dispatch/execution unit*. It is also responsible for the *out-of-order* execution capabilities of the processor. The micro ops are executed as their operands are available. A dependency analysis in the fetch/decode unit provides data integrity.

The results produced by the dispatch/execution unit have to be committed to the machine state by the *retirement unit*. In order to support data integrity with respect to speculative execution, speculatively executed micro ops may only be retired, after the path of the speculative executed code is confirmed. According to target the micro ops are retired in-order to the data cache or the CISC *Intel architecture register set*.

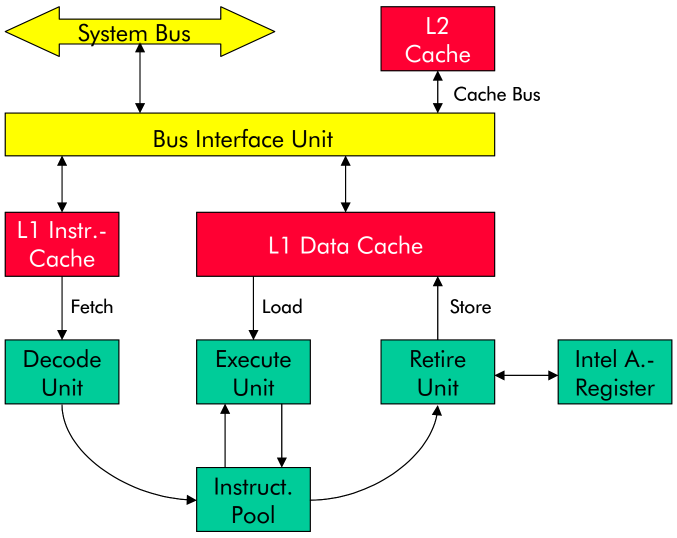


Figure 4.2: Building Blocks of the Intel P6 Architecture[41]

Due to the out-of-order execution unit, the measurement has to ensure that all instructions under test are executed within the measurement window. At the same time it should avoid the execution of unnecessary instructions within that same window. To ensure this serialising instructions have to be executed before the first, and after the last instruction under test. Whenever a debug register is written on the Intel P6 family such a serialisation takes place.

Within a target address type the unconditional and conditional branch instructions `JMP` and `Jxx` all share, with one exception, the same complexity and amount of memory needed for the arguments. Such target address types are for example *register indirect*, *far*, *near*, and *short* branches. The exceptional branch instruction is a short `JCXZ` opcode. The complexity of this opcode is larger, since it checks whether the register CX is zero and takes the branch if true. To ensure correct measurements in the case of branch opcodes, which have to be instrumented, one has to check if the instrumentation of such an opcode would be necessary. If this is the case, the opcode should be replaced in the assembler code by a work around like:

```
mov   %cx,%cx
je    $TARGET_LABEL
```

The feasibility of this change has to be checked as the case arises. It has to be stated that obviously this change must be maintained for the final executable as well. The impact of this restriction is rather low, since the `gcc` compiler used in the experiments seems to avoid this instruction by default. The data independent execution times of all instructions, especially multiplication and division, makes a further discussion of these instructions unnecessary. The processor is connected to the peripheral units via the GTL+ bus and the north bridge. The peripheral units are clocked with a lower frequency than the CPU. This induces a granularity in possible execution time probabilities. Additionally, the possible reordering of replies to requests on the GTL+ bus leads to further statistical influence on the execution time of a given piece of code.

4.3.2 Memory Organisation

The Intel P6 family provides the means for a paging mechanism of the processor. For this mechanism a fixed portion of memory can be mapped to any address used in the processor. This leads to the fact that the logical address of a program gives no indication of its physical location in main memory[1] or other storage devices. The caching mechanism on the other hand is dependent on the physical address. The usage of varying physical location memory with fixed logical addresses is very useful for general purpose operating systems, since this allows for efficient memory usage, but has to be avoided for real-time systems due to the loss in determinsism. Thus one has to generally ensure in real-time systems that the memory map is not changed from one execution to another. This will be further discussed in Section 4.5.1.3.

For the mapping of logical to physical addresses, which is done transparently for the software, the memory management unit uses a set of tables: the page global directory and the page tables. These themselves reside in main memory. The *translation lookaside buffer (TLB)* are little caches for these tables. To provide the worst case the TLB entries must be invalidated. This can be implemented by a write access to the register CR3, which is the base register indicating the start of the page global directory.

4.3.3 Caches

To close the speed gap between the fast CPU core and the comparably slow main memory, 2 levels of cache are deployed in the Intel Pentium III. The first level or L1 cache splits up into data and instruction cache which supports the processors harvard architecture. The second level or L2 cache is a unified cache. All caches utilise a 32 Byte cacheline. The exact cache sizes and their associativity must be individually checked for a processor with the `CPUID` opcode. The Intel Pentium III processor has an inclusive cache design i. e.; the data and code in the first cache level is also contained in the second level cache. For the tests the *write back* caching strategy (cf. Section A.1.3) is

[1]However, usually the page has to be aligned to a multiple of it's size. Thus in general the number of sets in the cache, the data may be located in is small.

assumed, thus modified data within the caches is written back on replacement of the corresponding cacheline. The replacement strategy for the caches is not clearly stated within the documentation. With an L1 cache a pseudo *least recently used (LRU)* algorithm is given, but the exact implementention of the "pseudo" prefix is not clarified further. The replacement scheme of the second level cache is not even mentioned.

4.3.4 Branch Prediction

The branch prediction is divided in three parts. The *return address stack (RAS)*, the dynamic and the static prediction scheme. The dynamic prediction scheme is based on the *branch target buffers (BTB)*. The BTB is a 16 way set associative cache of 512 entries that stores the target addresses of branch instructions. Whenever a branch instruction is reached, the branch prediction mechanism checks first whether the instruction has an entry in the BTB. If this is not the case, than the static prediction scheme is used. After the branch has been retired, the computed target is moved to a BTB entry. A pseudo random mechanism decides which BTB entry is displaced to make room for the new entry. Additionally, the prediction unit for this BTB entry is initialised depending on whether the branch was actually taken or not.

The prediction unit of a BTB consists of a 4 bit branch history shift register and sixteen 2 Bit bimodal counters with saturation. The shift register contains information on which of the last four encounters of this branch instruction have been taken, and addresses one of the counters. The counter is incremented up to a maximum value of 3 when the branch is taken and decremented down to a minimum value of 0 when the branch is not taken. If a valid BTB entry is found for a branch instruction the corresponding counter, as addressed by the shift register, is checked as to whether it is likely that the branch is taken, i. e.; the counter is set at 2 or 3, or not taken (the counter is 1 or 0).

After the branch is predicted the decoding of opcodes continues at the target address of the branch instruction when the branch is predicted *taken*, and at the succeeding instruction to the branch instruction when the branch is predicted to fall through (*not taken*). If the dynamic branch prediction scheme could be used, the target address is taken out of the BTB without further address computation.

The return address stack is a small hardware stack that stores the addresses of the succeeding instruction of the last 16 subroutine call levels. Whenever a `RET` is encountered, the top address on the RAS is taken and the decoding of opcodes is continued at this address.

As has been previously mentioned, no speculatively decoded and executed micro op of the instruction pool is retired, until the prediction is verified. In the case of a misprediction, which may be target or the *not taken* prediction, the speculatively micro ops are marked as unused and the decoding and execution is resumed at the correct address. The in-order retirement of micro ops allows only 40 subsequent micro ops to be pending. Thus a bound of affected branch instructions can be computed by using the information of produced micro ops per instruction given in Section 29 in [28].

4.3.5 Monitoring Support

The processor debug registers potentially provide the means for taking measurements. Up to four addresses can be monitored in that way. The *performance monitoring counters (PMC)* are two registers which were initially introduced as a means to optimise code. They can be programmed to count various events, without having any influence on the execution of the code. These events include, but are not limited to, the number of bus accesses, cache hits and misses, or the number of correctly predicted branches. In other words, they serve very well for the measurement of cache refill times which are needed for the use of penalties. They can be utilised to determine the amount of cache hits and correctly predicted branches within the measurement to compute the penalty for preemption. The PMCs support the building of the presented processor model and help to verify or rebute the manufacturers documentation. The timestamp counter is one of the model specific registers of the Intel P6 family. It is a 64 bit register which counts the CPU clock cycles since power on.

4.3.6 Relevant Facts for the WCET Measurement

As regards the previously described features of the processors three aspects can be identified which will then be seperately discussed:

1. How to ensure that the worst case path is measured?
2. How to provide the worst case state of the processor prior to the measurement and to show to what extent this is possible?
3. Where to set the measurement points and show which monitor implementation has been chosen?

In the following sections, the merit and drawbacks of different techniques is evaluated for use in our approach.

4.3.6.1 Path Enforcement and Detection

The first problem to solve is how to make sure that the worst case path is measured. Basically three approaches can be distinguished here:

1. Use symbolic execution to identify the input data creating the desired path for measurement.
2. Enforce the path to be measured by instrumenting the code.
3. Use random input data and detect which path has been taken during a measurement.
4. Use random input data and assume that the worst case situation is covered by utilising statistical methods.

The last alternative has been used, together with an evolutionary testing approach in [66], but experiments have shown that the underestimation of the WCET varies to a broad degree. Therefore, this approach has not been considered further. The third approach of identifying the taken path can be accomplished in two ways. The first would be by single stepping through the code, identifying the path and then measuring the WCET in one run. The second would be to add code for the identification. This could be, for example, incrementation of a given variable. Due to the inaccessibility of the instruction cache, a detection of a path by an external hardware monitor is impossible. The single stepping through the code is very time-consuming. The instrumentation for path detection and enforcement of this is equally complex. However, the first has the drawback that many tests are necessary to identify and provide a desired path, especially in the presence of factual dead code, i. e.; code which may only be syntactically reached.

The symbolic execution of code like in [22] and [19] has the advantage of additionally identifying the loop bounds, and detecting code which cannot be reached. The complexity of such a symbolic execution simulator, and the time needed for the symbolic execution of larger programs and the test bed generation, is considerable. The enforcement of paths can be done in several ways. The implementation has to allow for fall through and branch behaviour, and provide the control over the number of loop iterations. A fall through can be enforced as follows:

- replacement of the branch instruction with `NOP`s.
- replacement of the target with the address of the subsequent opcode.
- addition of code which sets or clears a flag in the status register.
- replacement of the conditional branch prediction with a more suitable one.

The replacement with `NOP`s to simulate the fall through has the drawback of additional cycles being needed for the decoding of the opcodes. A similar problem is induced by the introduction of additional code to enforce the flags in the status register. Thus, replacement of the target is the first choice for the enforcement of paths on an Intel P6 family processor.

In some cases the number of loop iterations has to be enforced. To achieve this, additional instrumentation code is necessary. Figure 4.3 shows the instrumentation of a loop controlling structure at the end of the loop body. The corresponding implementation for a control structure at the start of the loop is straightforward.

```
            Original                      Loop Forced
         MOV.L  $LCOUNT,cvar                       MOV.L  $LCOUNT,cvar
target:                          target:
           ⋮                                          ⋮
         NOP
         NOP
         NOP
         NOP
         NOP
         NOP                               DEC.L  cvar
         Jxx    target                     JNE    target
```

Figure 4.3: Iteration Number Enforcement for Loops

Prior to the loop body a counting variable (`cvar`) has to be initialised by a desired value which is in depicted case `LCOUNT`. Again the `NOP`s reserve the space needed for the decrementation of the counting variable. The penalty to be used for this is two cycles for every iteration of the loop.

For the purpose of the tests within this work the enforcement of paths has been chosen. The implementation of symbolic execution, which would be first choice in terms of accuracy and overhead, has been avoided due to its complexity and dependency on the source language. Furthermore, it is suspected that the restrictions necessary for the symbolic execution would be significant.

4.3.6.2 Worst Case State

Due to the fact that data independent execution time already exists on the Intel P6 family processors, the approach is freed from the task of providing the worst case data for the opcodes. In this way, the focus is set on the worst case state of the acceleration features of the processor. At the start of the measurement the caches have to be in the worst case state. For the data cache this is useless modified data in the cache assuming write back caching strategy (cf. Section A.1.3). Whenever a new cacheline for the application has to be loaded, a dummy cacheline of worthless data in the corresponding set has to be written back to main memory first, until there are no dummy cachelines left within the set. The instruction cache is much simpler and in a first solution it can be handeled in a similiar way to the data cache with write through strategy. Since the instruction cache cannot contain data which is changed, as compared to the corresponding memory region, no write back is necessary and useless unmodified data is sufficient. With the unified second level cache, the amount of data and instructions need to be considered. Since cache design is already included within the P6 family, at least the L1 instruction cache will reside unmodified in the L2 cache (cf. [62]).

If we are to be conservative, a penalty must be added to the measured values of first cache size write backs, since the possible location of first level instruction cache in second level cache is not fixed. A more realistic approach would assume that the amount of data not to be written back is low when compared to the pessimistic assumption that the cache is filled to 15/16 with data which has to be written back. In this way, this statistic deviation can be neglected. The only, partially, known replacement scheme presents a problem in itself. In order to cover this, more data than the size of second level and more code than the size of the L1 instruction cache must be loaded. Additionally, the load functionality has to be constructed so as to load the cache in random order. This is the only way that a sufficient randomisation can be assumed. To cover the described effects is to load first the L1 instruction cache and only then the data cache and L2 cache is filled. Another solution for the problem of instruction caches is to write “self modifying code”. A write access to a cacheline resident in the instruction cache causes this cacheline to be discarded out of the instruction cache. The entry is loaded in the data cache and tagged as modified. Since the write access to an address which is currently executed leads to a complete pipeline stall, the last couple of cachelines in the measurement routine can not be handeled that way.

The branch prediction is an additional problem which can not be provided in worst case due to the pseudo random replacement scheme of the branch target buffers, and the reset of the branch prediction at a new load of the branch target buffer. To provide at least a known state, a considerable number of branches including their targets have to be loaded into the branch target buffer. The reset of the branch prediction mechanism can be assumed the worst case, due to the fact that usually a previously adjusted branch prediction will enhance execution. A conservative approach would have to evaluate the worst case state here.

Another effect that must be covered is that an empty branch target buffer leads to the use of static prediction. It has to be checked, for each branch instruction in the code, whether the static prediction was correctly predicted for the first encounter of the branch opcode within a particular measurement. If the static prediction is correct, than a misprediction penalty has to be added for this instruction. In small loops more than one visit to the branch instruction might be necessary before the BTB entry is set. This is due to the effect that only retired branches influence the branch prediction and fill the branch target buffer when appropriate. Since it is only possible to have up to 40 micro ops in the instruction pool (the number of micro ops per CISC opcode is given in Section 29 in [28]) a bound of affected branch instructions can be computed. As the Intel documents deliver no bound on the misprediction delay, measurements were employed to provide this information. According to these measurements a misprediction incurrs up to 13 cycles (cf. Section 5.3.2).

4.3.6.3 Choosing the Monitor Technique

The first aspect which has to be covered is where to set the measurement points. In addition to the start and end of functionality, additional measurement points are often necessary due to structural problems caused by the decision to avoid the symbolic execution. Two reasons can be identified that necessitate the introduction of an additional measurement point. Firstly, code which has to be instrumented to enforce a path may not be reached more than once during one measurement, otherwise it would not be possible to measure one path in the first execution, and another in the second and so on. This holds true for subroutines with alternative paths which would otherwise be called more than once during a measurement, as well as alternative paths inside loop bodies. In the case of subroutines, a measurement point has to be introduced somewhere between the concurrent calls to the routine. User interaction could be useful here, to induce minimum overestimation by the additional measurement point. With loop bodies it was decided to place the measurement point just after the start of the loop. An exception to this rule is made, when the path of the loop body is known to be data independent. Typical examples of such data independent loops, with more than one possible path in the loop body, are the FFT butterfly algorithm, or the appliance of filters on images. In these cases no measurement point inside the loop is necessary.

The second reason for introducing an additional measurement point is the handling of complexity. Since all possible path combinations between two measurement points must be investigated, the number of paths which have to be measured increases rapidly. In order to cope with this, additional measurement points may be set. It is proposed, for efficiency sake, to concatenate measurements by designing the measurement routine in such a way that it integrates the taking of the measurement stop time stamp, the manipulation of the execution units and the taking of the first time stamp for the start of a new measurement.

The final decision for the monitor is how to set the measurement points. In the case of Intel Pentium III, three ways to achieve this seem possible. The first is to use the break points provided by the debug reigsters of the processor. The second method would be based on code replacement, similiar to the way it is done with debuggers. The third way to achieve measurements is to add a call in the assembler code.

The debug registers are programmed with the address to be monitored. A few additional bits define assertion conditions. Whenever the instruction at the address is decoded, as specified in the debug register, a breakpoint interrupt is generated. Debuggers often solve this differently, due to the fact that such hardware debug registers are not commonly available in processors. The access to the debug registers is only possible with operating system priority level on Intel P6 processors. In this case the instruction at the specified address is replaced either with a interrupt generating instruction, or with a call to a debugger routine. When the instruction is reached, the debugging routine replaces its breakpoint with the original code and moves the breakpoint to the succeeding instruction, executing the original code. After the second breakpoint is issued, the first breakpoint is restored and the code at the second breakpoint is replaced with the original. Thus all of the original code is executed while providing the access at a desired address of the program. Adding the call to a measurement routine in the assembler code is very simple. In order to provide correct behaviour for the final executable, the measurement routine has to be either replaced by a simple `RET` opcode, or the call has to be overwritten with a corresponding number of `NOP` opcodes.

The major drawback of the hardware debug registers is in the generation of an interrupt. The automatic mechanisms activated at an interrupt induce an undesireable overhead. With the debugger like instrumentation, either the small version utilising the `INT` opcode to generate an interrupt or the simple call of a measurement routine can be deployed. The first suffers the same problem as the use of a hardware debug interrupt, while the other version has the problem that the call to the measurement routine takes at least five bytes. When performing single stepping room for two of these calls is necessary. If the first branch instruction, after the intended point of measurement, lies within this first ten bytes the correct handling is troublesome. The single stepping would induce further overestimation on the execution time due to the necessary penalties. The addition of the call to the measurement routine in the assembler text, leads to the necessity of removing these calls for the final executable and thus also a overestimation is induced which is less then the one needed for single stepping. Thus the instrumentation of assembler code is chosen.

4.3.7 Preemption Handling

Within real-time systems preemptive real-time operating systems are often chosen. It has to be taken into account that the preemption of a task has an additional effect on the working set of the task, with regard to the acceleration techniques of a processor. Usually this prolongs the time the task utilises the processor i. e.; the computation time C of the task. In this discussion the term “task” indicates the code between two measurement points.

To handle preemptions, two possible approaches can be utilised. Either the preemptions are taken as a mere statistical effect on the execution time, or the interference on the acceleration units of the processor has to carefully analysed. While the first approach has the drawback of lack of determinism, it is nevertheless applicable under two conditions:

- The impact of the preemption has to be small as compared to the execution time of the preempted task.
- The pattern of preemptions, with regards to their interarrival time, must be at least pseudo periodic, i. e.; jitter and the occasional dropout of a preemption is acceptable.

These conditions are met if, for example, a long running application is periodically preempted by the clock tick interrupt. In such cases it is possible to circumvent the schedulability analysis for this particular kind of preemption by measuring the application in the preempted case. In this way the cost of preemption and the execution time of the preempting task, or interrupt service routine, is included in the measured execution time. In order to provide at least some confidence in the measured values, it has to be ensured that the start of measurement is not “aliased” with the preempting code, i. e.; the timely distance between start of the preempted task and the first preemption has to be varied from zero to a full “period” of the preemption. Since this method is only applicable for a small number of preemption scenarios the further discussion will focus on the detailed analysis.

In addition to the time for task switching itself, and the execution time of the preempting task, a schedulability analysis like the one described in Section 4.7 must take into account the prolonged execution time, due to the interference on the acceleration units of the processor. This prolongation can either be determined by modelling the usage of the acceleration techniques during the execution, or by simulation of the preemption.

To facilitate modelling of the usage of the acceleration techniques, a more simple method can be chosen than is used for the exact simulation of the processing hardware necessary for the estimation of the WCET. The maximum amount of cachelines, in the instruction cache, used by a piece of code can easily be computed. The bound on the amount of data cachelines must be derived from a number of components. The first component is the cache usage due to the usage of the stack. The stack is used for the storage of local variables, the return address and arguments of subroutine call, and to save registers which are used in the subroutine. In between two measurement points the stack usage can be simply expressed in cachelines. Nested subroutine calls are simply added up. Figure 4.4 depicts a sample code which is analysed according to the data cache usage in Figure 4.5.

To facilitate the top level routine, which contains start or end point of the measurement, the local variables on the stack have to be covered as global variables. Global variables

```
1 void sampleroutine()
2 {
3 char readbuf[BUFSIZE], writebuf[2*BUFSIZE];
4 /* get the data out of the input fifo */
5 rtf_get(IFIFO, &readbuf, sizeof(readbuf));
6 /* encode data */
7 conv_encode(readbuf, writebuf, BUFSIZE);
8 /* output data blocks to output fifo */
9 rtf_put(OFIFO, &writebuf, sizeof(writebuf));
10 return;
11 }
```

Figure 4.4: Sample Code for Figure 4.5

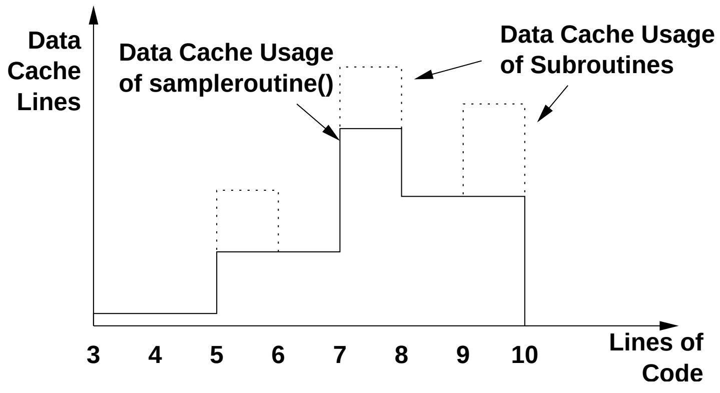


Figure 4.5: Data Cache Usage of the Sample Code in Figure 4.4

can be regarded as having a live span within the two measurement points marked by the first and last access of the variable. Thus a bound on the amount of valid useful data cachelines can be given at any given point within this interval. Correspondingly the usage of the TLBs can be conducted.

The usage of branch prediction mechanims can only be investigated with regard to loops. A small simulation of the branch prediction mechanism provides information on how many iterations are necessary until the dynamic branch prediction mechanism is *tuned* to a particular branch instruction in the loop. The best case has to be assumed for preemption of the BTB entries. As regards the static branch prediction scheme, it is now possible to bound the number of correctly predicted branches which might be turned into mispredicted branches.

The simulation of preemptions itself can either investigate the actual prolongation of the execution time, or the interference on the acceleration techniques. Due to the inaccessability of instrumenting the code to be "preempted" at a particular loop iteration for a systematic testing, only a statistic external induced preemption is possible. To gain statistical confidence for this approach a multiple of the number of measurements necessary for the WCET estimations would be necessary. Thus this approach is not followed further.

To gain information regarding the interference utilising measurements, an additional measurement point can be shifted slowly from instruction to instruction, measuring the location of the measurement point with every change. With these measurements not the actual time, but the amount of cache hits and correctly predicted branches is of relevance. The branch prediction mechanism have only to be taken into account in loops, which will be seperately discussed. With a code which is not a loop between two measurement points, the effects of the branch prediction are already covered in the WCET. In this section the term *cache hits* refers to L1 instruction and data cache. L2 cache hits will be handled seperately. The sum of the measured cache hits, before and after the additional measurement point, is compared with the uninterrupted case. The correctly predicted branches are investigated accordingly. Due to the limitation on two performance monitoring counters, twice the number of measurements are necessary to gather the relevant information on cache hits. On the other hand only a fraction of the measurements for the WCET is necessary, due to the lesser variance of these values. Since only a very limited number of TLB entries is usually used, a bound on the misses can be taken by utilising the TLB misses measured in the uninterrupted case.

Loops again need special treatment. Since it is impossible to place a measurement point inside the loop, which is executed only at a given iteration without adding a significant amount of code, a bounding of the impact of preemption is used. This special treatment is obviously only necessary for those loops whose loop bodies are not already instrumented with a measurement point. Instead of only one, two break points are set. One directly before the loop and one behind it. Thus the original measurement is split into three parts. For the remaining cache hits H with preemption within the loop the following equation is used:

$$H = \min(H_1) + \min(H_2) - \left\lceil \frac{\max(H_2)}{N-1} \right\rceil + \min(H_3) \tag{4.6}$$

where H_1, H_2 and H_3 are the cache hits for area 1, 2 and 3 respectively and N is the number of loop iterations. It is assumed that the first iteration of the loop produces no cache hits. This is a conservative approach since, due to the long cachelines, even serial code usually produces cache hits. However, this can not be taken as the only penalty, since there may be caching effects which reach from outside into the loop body, or even beyond the loop. The two measurement points before and after the loop ensure that these effects are covered.

With branch prediction mechanism only the second area, i. e.; the loop, is investigated as regards the mispredicted branches, and not the correctly predicted branches. Since the branch prediction is initially “ignorant” of the loop, a preemption within the loop will cause in worst case as many additional branch mispredictions as in the initially measured case due to its renewed “ignorance”. Additionally it has to be noted that the up to 16 additional branch mispredictions have to be taken into account, depending on the maximum nesting level of routines, to consider the effects of the return address stack.

The final question is how to set the additional measurement point. Again the two alternative methods of utilising the debug registers and inserting code have to be considered. The introduction of additional code would only be possible using the debugger style instrumentation, with single stepping over the code, since all points in the code are to be investigated. On the other hand the usage of the hardware supported breakpoints by utilising the debugging registers has lost its drawback of adding uncertainty to the execution time, since only the number of cache hits and the branch prediction is of relevance. Due to the much easier implementation and lesser overhead during the measurement, the usage of the hardware provided debug mechanism is favoured.

4.4 AMD Athlon Family

The architecture of the AMD Athlon processor family has, on first sight, many similarities to Intel’s P6 family, however, in detail it differs significantly. After a brief description of the processor features, the discussion of the measurement technique is focused on comparing the differences to the measurement of software on Intel Pentium III processors. As with the Intel processors, the Sections describing the processor are almost free of references. A much more detailed description of the AMD Athlon family can be found in the corresponding Sections in Appendix A.2.

4.4.1 General Architecture

The basic architecture depicted in Figure 4.6 is similar to that of the Intel Pentium III. The core executes like Intel’s counterpart RISC code. The three parallel decoders are, unlike the Pentium III, identical. Complex instructions are fed into a micro code sequencer called *vector path*, while simple instructions are directly decoded *(direct path)*. The 9 way execution core also provides out-of-order execution. Thus for the measurements on the AMD Athlon a serialising opcode has to be used, to ensure the complete execution of all relevant opcodes within th measurement interval. The conditional and unconditional jump opcodes all share the same complexity as regards decoder usage and space with one unfortunate exception. Similar to Intel’s Pentium III, the short branch opcode `JCXZ` is more complex than all other instructions. The `JCXZ` opcode utilises the vector path, instead of the direct path of the short `JMP` and conditional `Jxx` opcode. The impact of the previously described workaround is rather low. Furthermore, the AMD Athlon optimisation guide [4] states that vector path instructions should be avoided and the `gcc` compiler used for the experiments makes no use of this instruction.

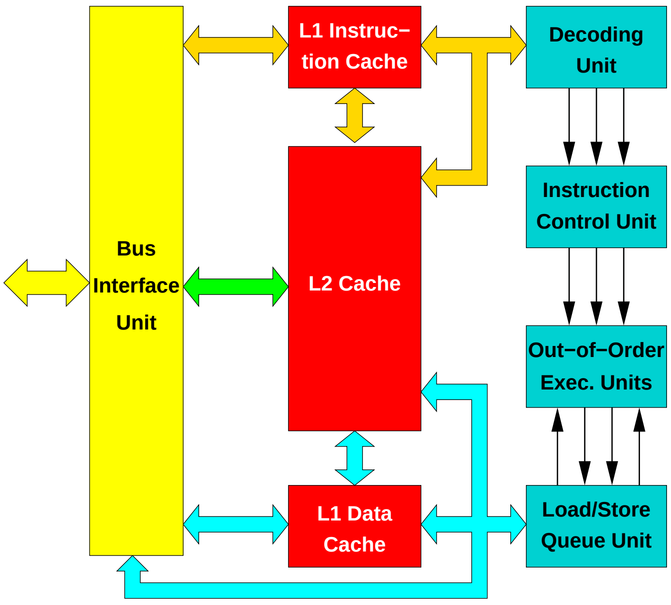


Figure 4.6: Building Blocks of the AMD Athlon Architecture

Fortunately, the Athlon implements the multiplication and division in a way that the execution time of these instructions is data independent as well. Connection of the AMD Athlon to the other system components is implemented by the EV6 bus. The concept of outstanding transactions of the EV6 bus can be compared to the Intel counterpart, which cannot be induced at measurement time and therefore add to the statistical property of the execution time. Similar to the Intel P6 family, the lower clocking frequency of the peripheral leads to quantisation of delays.

4.4.2 Memory Organisation

The memory access possibilities like addressing modes and paging mechanisms, are almost identical to Intel's. The major difference is the number of TLBs available. To create a worst case scenario for the memory accesses all TLBs have to be invalidated. As with the Intel P6, a write to register CR3 is sufficient to provide this.

4.4.3 Caches

The cache architecture of the AMD Athlon is, like Intel's, divided into two levels and in the first level further split into data and instruction cache. The exact size of the caches for a given processor have to be determined by utilising the `CPUID` opcode. In contrast to the Intel P6, the Athlon has an exclusive cache design, i. e.; data or code contained in the first level cache is not present in the second level cache. In order to provide a fast mechanism for an L1 cache load from L2 cache in the case of a full set, the victim buffer is introduced. The victim buffer can take up to 8 Cachelines displaced from the L1 cache, in order to make room for other cachelines loaded into the L1 caches. In the interest of efficiency this displacement is done in parallel to the load. On idle cycles of the victim buffer it is drained in the background to the L2 cache. In the case of a full victim buffer it is written entirely back to the L2 cache. This case induces a 18 cycle delay until the new cacheline is loaded into L1. Another central difference to the Intel P6 family is the 64 Byte cacheline which is twice the size. Additionally, the AMD provides implicit prefetching, i. e.; whenever a cacheline is loaded into L1 instruction cache, the subsequent cacheline is loaded as well. Thus a higher hit rate for the instruction cache is achieved.

4.4.4 Branch Prediction

The branch prediction of the AMD Athlon is like that of the Intel P6 family, divided in three parts: *return stack buffer (RSB)*, static and dynamic prediction scheme. The RSB corresponds to Intel's RAS but is, with 12 entries, slightly smaller.

The dynamic branch prediction is further split into four parts:

1. predecode cache
2. branch target address cache (BTAC).
3. global branch history (GBH).
4. 2048 global history bimodal counter (GHBC).

Alongside the L1 instruction cache, the predecode cache is stored and contains additional information like identified branch instructions and instruction boundaries. The branch target address cache is equivalent to Intel's BTB, while the GBH is a 8 Bit shift register used to note down whether the last 8 branches where taken or not. The GBH utilises the fact that branch instruction close together often follow a correlated systematic behaviour (cf. [23]). The GHBCs are equivalent to Intel's 2 Bit bimodal counter with saturation. In contrast to Intel's version, the GHBCs and the shift register are not associated with a particular BTAC. As the name implies, the GBH contains the result of the 8 branch instructions, regardless of their address. The GHBC are organised in 8 rows and 256 columns. While the row taken for prediction depends on the address of the branch instruction, the column is chosen by the GBH. A speciality of the AMD Athlon is that the target address of short branches is computed on the fly, and the dynamic prediction by the GHBC is always used for these short branches. Near and non control transfer far branches use the static prediction rules when no valid BTAC entry for the branch instructions exists. Far control transfer branches are always statically predicted. The "global" memory of GHBC and GBH is a problem for providing the worst case during measurement (cf. Section 4.4.6.1).

4.4.5 Monitoring Support

The major difference to Intel P6, as regards the monitoring support, are the 4 PMCs provided by the Athlon. With these, a slightly more detailed view of the execution is possible. In addition, the Athlon provides the 64 Bit wide time stamp counter register. The debug registers are identical to Intel's.

4.4.6 Relevant Facts for the WCET Measurement

Due to the structural similarities between the Intel P6 family and the AMD Athlon, the techniques for measurement are roughly the same, while differing in detail. The following discussion will focus on the differences in technique when compared to those used on the Intel Pentium III (cf. Section 4.3.6).

4.4.6.1 Path Enforcement and Detection

Due to the complexity of the approach, symbolic execution is also avoided on the AMD Athlon processor. The detection of paths would, as with the Intel processor, depend on additional code and it would take an enormous amount of time to test the right input data combinations.

The replacement of branch instruction by `NOP`s, and the replacement of the branch target address, have both the drawback that the branch prediction mechanism is not utilised correctly. Since short branches always utilise the dynamic branch prediction, and this has a global "memory" of taken and not taken branches due to the global branch history and the global history bimodal counters, the usage of the described instrumentation would lead, at the very least, to overly pessimistic results. In the presence of uninstrumented loops security margins would have to be added to the measured results in order to provide a secure result due to the absence of *not taken* branches. Uninstrumented loops are possible whenever the loop body is data independent, and the number of iterations fixed. A possible alternative would be to replace the code in the following way:

```
      Original                         Force Branch
                         taken                          not taken
NOP
NOP                 MOV.L  %esp,%esp        MOV.L  %esp,%esp
Jxx  target         JNE    target           JE     target
```

The two `NOP` opcodes, which are inserted in the assembler text, allow for later replacement during measurement with the `MOV` opcode. The usage of the stack pointer `%esp` for manipulation ensures that, after this move the zero flag is cleared. For each of these instrumentations a 1 cycle penalty has to be introduced for the additional opcode to be decoded. For measurement points inside loop bodies, it has to be checked whether the static prediction is used. On branch instructions where this is the case, with the static prediction potentially yielding better results than the dynamic prediction, a number of misprediction penalties need to be added to the execution time bound of the loop. Fortunately, the initial condition is not valid for short jumps which are usually used to a greater degree than near or far branch instructions.

4.4.6.2 Worst Case State

The handling of caches and TLBs is identical to the Intel version with the exception that, due to *least recently used* policy, it is sufficient to use L1 instruction cache size code, and L1 data cache plus L2 cache size modified data, to displace all useful code and data in the cache.

To perform the branch prediction, a different approach must be followed. The analysis necessary to determine the worst case state for a particular piece of code, with respect to the path to be measured, is very time consuming and often leads to ambiguous results, due to the open decision on where to place a potential correct prediction.

A conservative approach, utilising a small but exact model, would assume the 8 starting instructions as mispredicted, until the state of the GBH is known. After that the model would start to define the GHBC states by removing those states an individual GHBC is definitely not in. The following example shall demonstrate how this could be done. After a taken branch, the corresponding GHBC might be in state 1, 2 or 3 but not in state 0. Assuming that when this particular GHBC is next visited, and the branch is taken once again, the GHBC must be in state 2 or 3. Thus, the next time the GHBC is utilised it will predict a branch *taken*. After a considerable interval, all GHBC states would be known, but one would have assumed so much mispredictions that the WCET would be better off with a static prediction like, for example, always *taken*. An additional uncertainty introduced to this approach would be the out-of-order execution. The time penalty for a misprediction has been measured to be 13 cycles (cf. Section 5.3.1). A correctly predicted branch incurs no penalty for short branches, and a 1 cycle penalty for all other branch instructions handled by the branch prediction mechanism. Due to this, the statistical approach has to be used. A considerable amount of measurements are done, preferably with as many different start conditions as possible. In order to ensure this, random jumps have to be executed prior to the start of each measurement. Only then can a random state of GBH and GHBCs be assumed.

The necessity for a load of caches with useless code and data, and the randomisation of the branch prediction seem contradictory. The filling of the instruction cache may not be done after the filling of the data cache, in order to ensure that the complete L2 cache is filled with data, and not with code displaced out of the L1 instruction cache. On the other hand, the usage of loops in order to load the data into the caches would determine at least partially the state of the dynamic branch prediction. The solution lies in the integration of all three tasks in a monolithic piece of code. This code has to load and "modify" the data in the caches, randomise the dynamic branch prediction and be exactly the size of the L1 instruction cache. Additionally, code already executed is also "modified", thus ensuring that the L2 is filled with modified data. The "modification" is achieved by adding a 0 to the "data" located at that address.

4.4.6.3 Choosing the Monitor Technique

The options for introducing additional measurement points, and the method of implementing the measurements on AMD Athlon processors, are identical to those of the Intel P6 family as described in Section 4.3.6.3. Therefore, we also choose the addition of a call to the measurement routine in the assembler code.

4.4.7 Preemption Handling

The method of shifting a measurement point through the code, in order to "simulate" the effects of preemption and the utilisation of the hardware debug mechanism, is also possible for the AMD Athlon. The duplication of performance monitoring counters allows us to measure all relevant monitoring events in one run. The caches and return address stack can be handled according to the mechanisms already explained in Section 4.3.7 for the Intel P6 family processors.

Again with loops, two measurement points are set directly before and after the loop. Due to the properties of short branch instructions, utilising the dynamic prediction scheme, all three areas have to be investigated. Firstly, the number of mispredictions of all three parts are summed up. In order to provide a bound for the mispredictions inside the loop the misprediction in area 2 has to be added twice, thus "simulating" the ignorant case after an preemption within the loop. The resulting value is compared to the mispredictions of the uninterrupted case. Equation 4.7 shows the number of mispredictions M which need to be taken into account for the branch prediction related preemption penalty within a loop. M_1, M_2, M_3 and $M_{nopreempt}$ respectively express the number of mispredictions measured for the three parts, and the "original", unpreempted case.

$$M = \max(M_1) + \max(M_2) + \max(M_3) - \min(M_{nopreempt}) \tag{4.7}$$

This number of mispredictions has to be taken into account for a preemption within the loop.

4.5 Interaction with a Real-Time Operating System

In this context, the first step is to define the difference between a general purpose operating system *(GPOS)*, and a real-time operating system *(RTOS)*. A RTOS is usually focused on providing deterministic behavior, while the GPOS provides more flexibility which is only sparesely utilised in a RTOS. Typical examples of this flexibility are the dynamic memory management, or the generation of additional tasks at run-time. The following describes a few examples which show the similarities and differences between RTOS and GPOS.

As with standard operating systems, the major purpose of the RTOS is to manage access to the resources of the system. The prime resource in a computer system is the resource CPU i. e.; the time alloted to a task for execution. This is handled by the *scheduler* of the operating system.

With the scheduler we have the first difference between a GPOS and a RTOS. While the scheduler of a GPOS often tries to split the computation time evenly, or at least fairly under the demanding tasks, the focus of an RTOS lies on determinism and meeting the deadlines. Examples of GPOS scheduling policies are *round robin*, *dynamic priorities* or even *cooperative* and *first come first serve* scheduling, while many RTOS provide scheduling algorithms like *fixed priorities*, *earliest deadline first (EDF)* scheduling and *time slicing* to name a few. To list and explain all scheduling policies proposed and implemented would exceed the scope of this work, and therefore, further discussion will be limited to fixed priority scheduling, due to the relevance of this operating system in our experiments.

Other resources which need to be managed are, for example, memory and peripheral units (e.g. timers). Some of these resources are handled by manager parts inside the operating system, others are managed using mutual exclusion techniques like semaphores.

RTOS kernels are relatively small. One reason for this is the frequent deployment in small sized embedded systems, another is that a GPOS usually provides additional dynamic functionality like dynamic task creation or swapping memory to disk. A deeper discussion of real–time operating systems and their properties can be found in Section 10 of [48].

As regards the real-time operating system, two major aspects must be addressed for a complete WCET approach:

1. The WCET of the RTOS routines itself, with respect to their interoperability with other RTOS routines and application software.
2. The impact on the execution times of the application software as induced by preemption.

The following section describes a special operating system case for real-time systems, namely, a GPOS with real-time extensions. Following this, diverse components of the RTOS are addressed, while the last two sections describe the impact of preemption on the WCET of a thread, and the thread model used for the real-time analysis.

4.5.1 Extensions to a General Purpose Operating System

As previously stated in Section 4.2, the provided support for numerous hardware components, and the well known user interface are the main reasons for using a general purpose operating system with real-time extensions. However, other reasons can be named as well: The development environment is well known by software developers, and is usually either cheaper or of better quality than those available for a given RTOS. Due to this, hardware and software upgrades can be managed easily.

To suit the purpose of the presented work, Linux with RT-Linux as real-time extension, has been chosen as an exemplary work. The reason for this, is the availability of the sourceode of both, the GPOS sourcecode, and the RTOS extension. The source code of commercial GPOS Microsoft Windows or Solaris is almost impossible to get access to. By utilising the sourcecode a deeper insight into the mechanisms is possible. Since Linux has only a marginal impact on the method itself, only the essential elements will be described as thy are needed. Too see a more detailed view of this free operating system see e. g. [9], [85], [93] and [94].

The usual way to extend a GPOS by RTOS functionality is to insert a tiny layer between the hardware and the GPOS, and add the RTOS functionality in parallel to the existing operating system. Figure 4.7 shows how this is accomplished with Linux and RT–Linux.

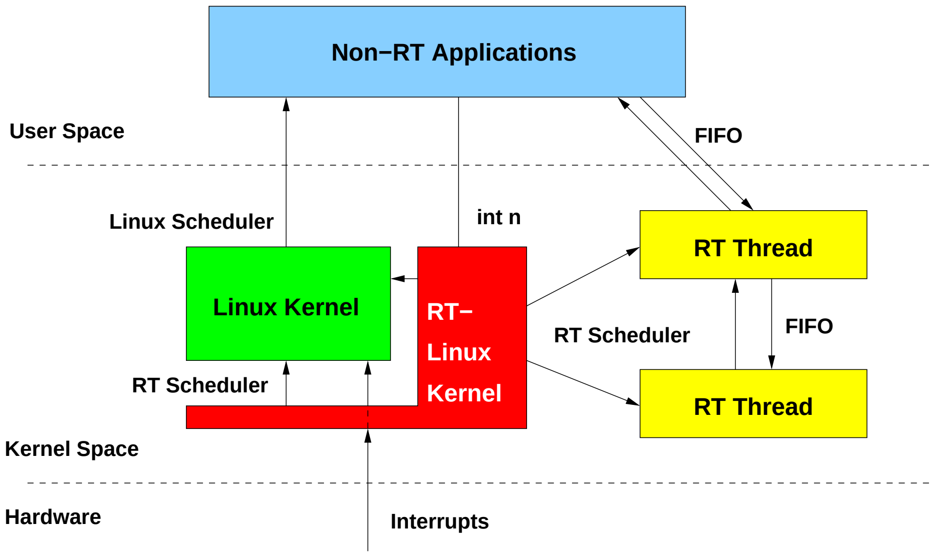


Figure 4.7: Interaction between Linux and RT–Linux Kernel [81]

The use of extensions to general purpose operating systems has its disadvantages as well. The most important impact is the time needed to switch from user mode tasks into the kernel mode. This change in the privilege level induces an considerable additional latency of interrupts. In addition, the GPOS produces a serious interrupt load, and could initiate DMA transfers of peripheral units, like the harddisk and network interface.

4.5.1.1 Thread Management

Application software, in general purpose operating systems, is mostly organised in *tasks*. Each task has its own memory space as demonstrated in the segmented memory model, described in Appendix A.1.2. Thus, it is not possible for a task to write directly into the memory space of another, with the exception of *shared memory*. Communication between tasks is realised using *system calls* to the operating system. Additionally, application tasks are usually placed at a privilege level below that of the operating system. Thus, the system is protected against faults in the user tasks, and resource management is handled exclusively by the operating system.

However, the neat separation of tasks with memory protection has its price. To initiate a system call and an interrupt request, the privilege level and the memory segments have to be switched with every entering kernel or task code. The time needed for the transition of privilege level and memory space is consuming and varies often to a broad degree. On and AMD Athlon system with 900 MHz a value between 20 to 25 μs has been measured (cf. [81]).

To avoid this, and the induced latency of interrupts, real-time operating systems are usually organised in threads. A thread is a light weighted task which operates in the same memory space, and privilege level, as the other threads. In an RTOS, these threads frequently share the memory space and privilege level with the operating system kernel. Thus, thread switching times and interrupt latency are kept to a minimum. The omitted protection is justified with the assumption that the system is verified or tested thoroughly before deliverance. A programmer of RT threads must ensure that he uses the correct way of addressing resources, as it is quite easy to circumvent the resource management of the operating system. Some RTOS kernels like, for example, QNX or VxWorks provide additional full tasking support, but this is usually avoided in favor of the threading implementation.

In RT-Linux the threads work priority controlled on the same privilege level as the operating system. The Linux kernel is the thread with the least priority, and may execute when no other RT-thread needs the CPU. Linux may then use the conventional mechanisms to work with its user applications. Thus, for the RT-Linux kernel, the interrupt latency, due to the privilege and memory space transition, is still valid and has to be kept in mind.

4.5.1.2 Interrupt Management

The basic concept in the RT-Linux implementation of real-time extension is to treat Linux as an idle task of the RTOS. To implement this, the interrupt management is completely intercepted by RT–Linux. This includes, not only the interrupt handlers, but also the timer management. Since time is the essential resource in real-time systems, the timer is controlled by RT-Linux. The mechanism used for incoming *interrupt requests (IRQ)* is shown in Figure 4.8.

Hardware IRQs are interupts from peripheral units. Examples of such units would be the keyboard, hard disc controller, PCI device or the timer. All these are collected in the *programmable interrupt controller (PIC)* of the PC, which has been described in Section 4.2.2.3. The job of this PIC is to trigger the IRQ pin at the processor. Software IRQs are special instructions placed in the code to initiate an interrupt request. They are referred to as *traps*, or *synchronous interrupts* since they are generated whenever the processor executes a particular piece of code. These are often used to initiate a system call to the operating system, in cases where the operating system runs with a different privilege level. The third group of IRQ sources are *exceptions*. Exceptions are generated on a system fault. Examples of such system faults would be division by zero, or a segmentation violation.

Whenever an interrupt occurs, the interrupt gate of RT-Linux is called. This hands the interrupt down to the routine `rtl_intercept()`, which in turn checks whether a given

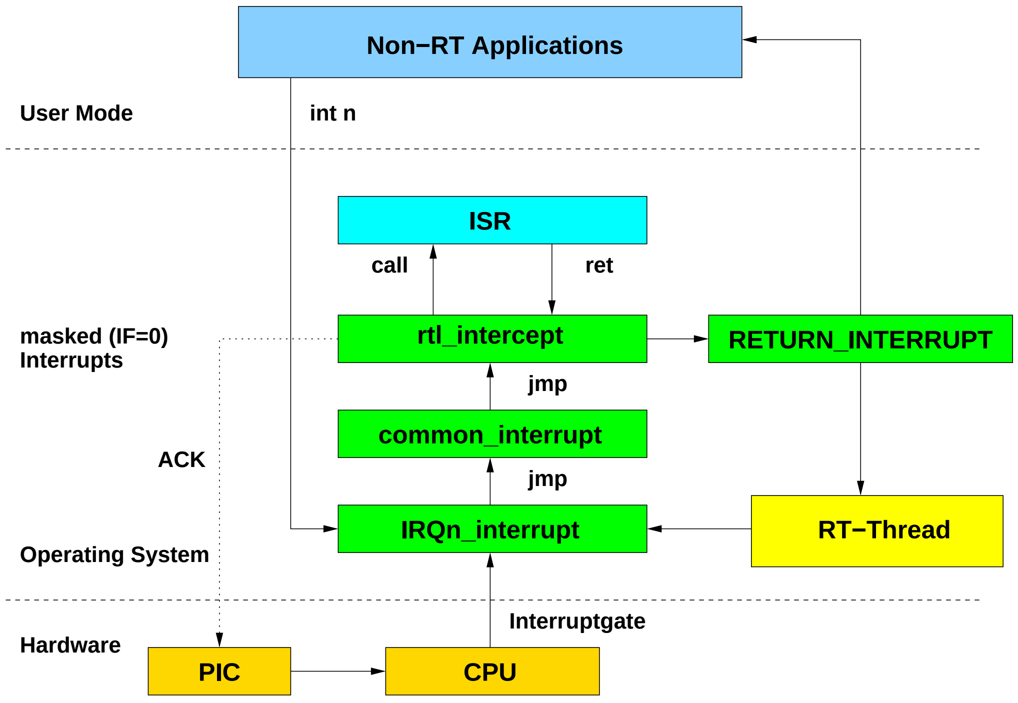


Figure 4.8: Interrupt Handling in the RT–Linux Kernel [81]

interrupt is relevant for RT-Linux. If relevance is proved, `rtl_intercept()` calls the corresponding *interrupt service routine (ISR)*. If the IRQ should be handled by the Linux operating system a flag is set in a variable, which indicates that this particular IRQ has occurred and the real-time operation is continued. Whenever the Linux operating system resumes execution, it checks the variable as to whether there have been interrupts which need be handled. If so, it executes the corresponding ISR. Due to the nature of the variable, multiple interrupts of the same kind are evaluated only once, therefore interrupts may be lost for the Linux operating system.

4.5.1.3 Resource Management

Resources within an operating system are often managed using device driver software. This driver concept provides a uniform interface for the application programmer and controls access to the resource. Direct access of the resource, by means of a thread utilising memory mapped I/O, is usually avoided in operating systems and only provided when the overhead of the driver is unacceptable. The handling of resources bearing the following two aspects in mind:

1. Determinism of the access.
2. Mutual exclusion.

The determinism of a resource access may be influenced by software and/or hardware. Different execution paths within the software of the resource driver are the main reason for the indeterminism induced, while latencies are usually responsible for hardware indeterminism. An exception to this is the memory management. This important resource has effects beyond simple latencies. The placement of code and data is critical to ensure reproducible results for the measurement. Unfortunately, the real-time extension RT-Linux utilises the memory management of the Linux GPOS. Due to the module concept of RT-Linux, in combination with the dynamic memory management of the Linux operating system, the location of the RT-Linux operating system and the real-time threads in memory are not fixed. To circumvent this problem, the module loader of Linux was modified to use a memory area, reserved at boot time, to allocate the memory for the modules. In this way, the location of the real-time portion of the code, i. e.; RT-Linux, and the RT-threads in memory are fixed, provided that the modules are loaded in a fixed order.

The software induced indeterminism has to be covered in the same way as other system calls, as described in Section 4.5.2. With hardware related indeterminism the difference between the maximum and minimum latency has to be found according to the process described in Section 4.2.2, and added to the execution time for every access to this resource.

An additional significant issue for device drivers is mutual exclusive access to a resource. The concurrent access to shared data is a typical example where using mutual exclusive access is used. There are two situations where a *data set* in this shared memory has to be protected against access by another thread.

- the data set is to large to be written within a single memory access
- the data set has to be accessed in a consistant *read, modify, write* manner

A common method used to realise this mutual exclusion is by means of *semaphores*. A thread may only access a resource when it holds the corresponding semaphore which may be obtained by a system call. The system call returns when the semaphore was successfully obtained. When the semaphore is held by another thread the system call is blocked. The core of this obtain operation is *atomic* i. e.; it is guarded against interrupts. As long as the thread holds the semaphore, no other thread is able to obtain it. After the thread has finished the transaction on data guarded by the semaphore, it releases the semaphore.

The usage of mutual exclusion mechanisms can induce the potential problems of *dead lock* and *priority inversion*. While the first is resolvable with intelligent programming, the second can not be avoided by means of programming rules. A simple deadlock scenario may occur when two threads use two or more resources concurrently, but request the semaphores in different order. Figure 4.9 depicts a simple example of such a situation. Thread τ_i requests the semaphore S_1, prior to semaphore S_2 while thread τ_j does the same in reverse order. After each of the threads has obtained one semaphore, the other semaphore is unavailable.

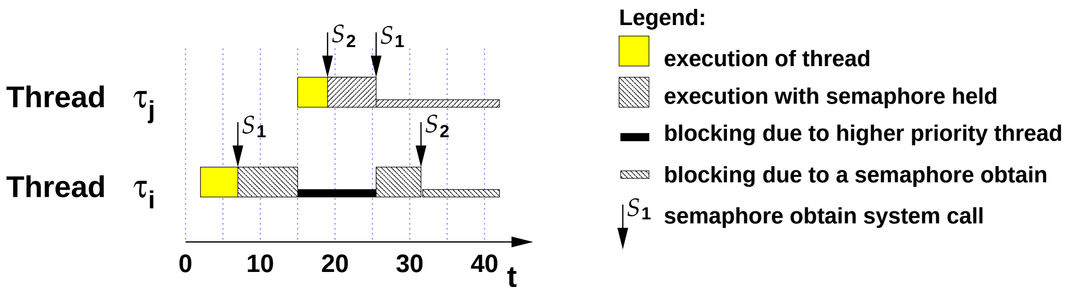

Thread τj
Thread τi
S2
S1
S1
S2
0
10
20
30
40
t
Legend:
execution of thread
execution with semaphore held
blocking due to higher priority thread
blocking due to a semaphore obtain
S1 semaphore obtain system call

Figure 4.9: Example of a Deadlock induced by the Use of Semaphores

As previously indicated, this problem may be circumvented by statically analysing the semaphore usage of all threads, and changing the order in which the semaphores are requested within the threads.

A more complex problem is the priority inversion, as depicted in Figure 4.10. As described before, a thread τ_j may be blocked due to a semaphore S_1, which is held by thread τ_i. In the case of an intermediary thread τ_k, thread τ_j is blocked longer than necessary by a thread of lower priority than itself. This case is called *priority inversion*.

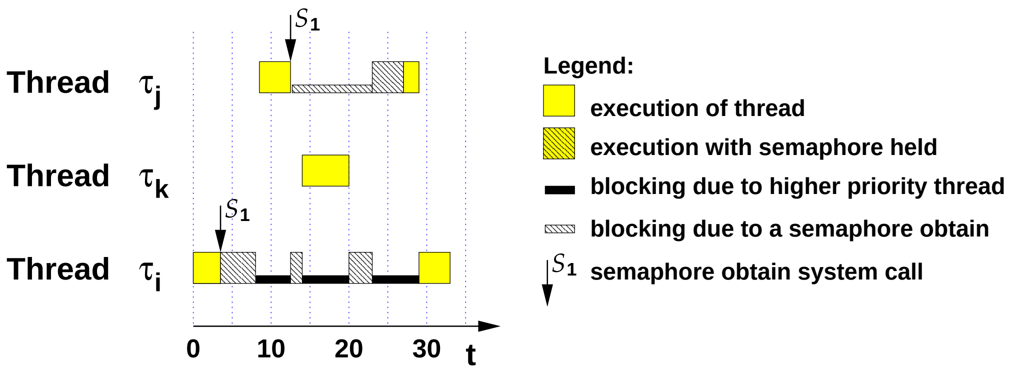
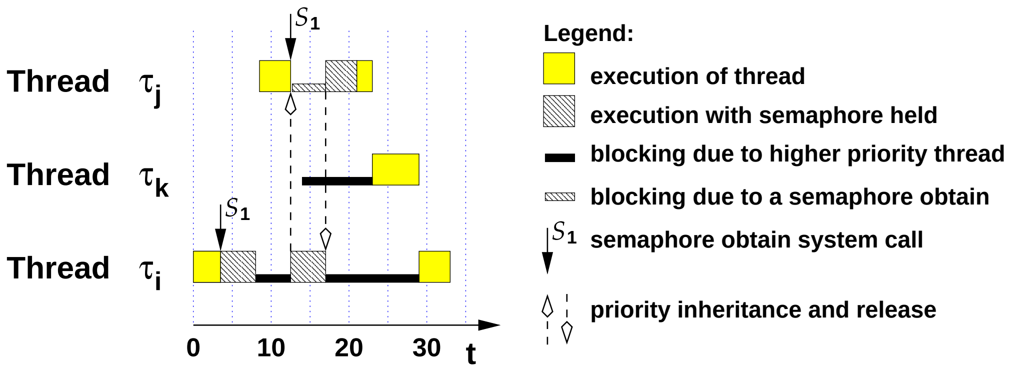

Thread τj
Thread τk
Thread τi
S1
S1
0
10
20
30
t
Legend:
execution of thread
execution with semaphore held
blocking due to higher priority thread
blocking due to a semaphore obtain
S1 semaphore obtain system call

Figure 4.10: Example of Priority Inversion

In order to solve this a *priority inheritance protocol* may be used. When thread τ_j is blocked due to a semaphore S_1, held by thread τ_i, the priority of thread τ_i is raised to thread τ_j. Thus an intermediary thread τ_k may not prolong the response of thread τ_j. Figure 4.11 shows a sample gantt diagram.



Thread τ_j
Thread τ_k
Thread τ_i
S_1
S_1
0
10
20
30
t
Legend:
execution of thread
execution with semaphore held
blocking due to higher priority thread
blocking due to a semaphore obtain
S_1 semaphore obtain system call
priority inheritance and release

Figure 4.11: Priority Inversion Avoidance by a Priority Inheritance Protocol

In this way, the blocking time B_{i,S_m} of a thread τ_i, trying to obtain a semaphore S_m may be bound provided the maximum execution time H_k, in which a given thread τ_k holds a semaphore, is known. The Equation 4.8 describes how the the blocking time B_{i,S_m} is determined. In this simple example $\mathbf{P}_{S_m}$ denotes the set of threads, which require the semaphore S_m during their execution, $\mathbf{L}_i$, the set of threads with lower priority than thread τ_i and $\max(\ldots)$, the maximum value of its arguments.

$$B_{i,S_m} = \max\left(\{H_k : \tau_k \in \mathbf{P}_{S_m} \cap \mathbf{L}_i\}\right) \tag{4.8}$$

This blocking time is also valid for all threads thread τ_i, which utilises not semaphore guarded resource S_m itself, when a thread of higher priority thread τ_j may be blocked by a thread of lower priority than thread thread τ_i. Figure 4.12 shows an example for such an indirect blocking. With cases of multiple semaphores used within a thread, the analysis gets considerably more complex. A detailed analysis model used to compute the blocking time, B_i, is presented in Appendix A in [73].

The priority inheritance protocol is not yet implemented in RT-Linux but announced as one of the next releases. Further information regarding priority inheritance protocols may be found in [73], and [74].

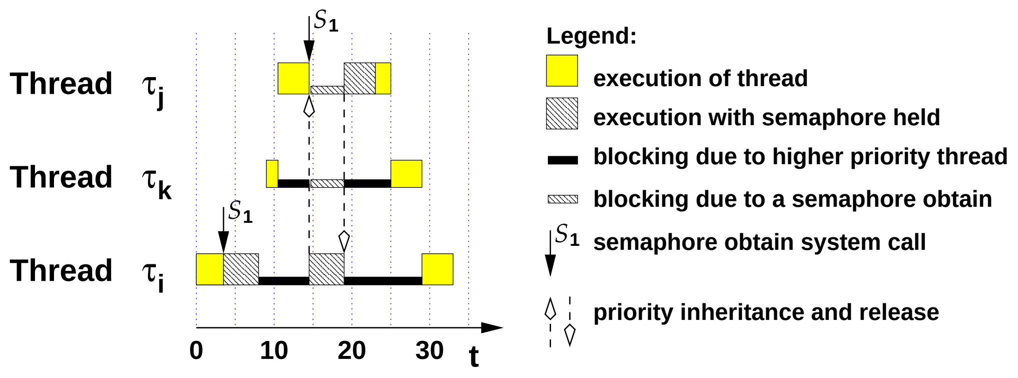

Thread τ_j
Thread τ_k
Thread τ_i
S_1
S_1
0
10
20
30
t
Legend:
execution of thread
execution with semaphore held
blocking due to higher priority thread
blocking due to a semaphore obtain
S_1 semaphore obtain system call
priority inheritance and release

Figure 4.12: Example of Indirect Blocking of a Thread

4.5.2 System Calls

Services provided by the operating system are usually utilised by the application threads via *system calls*. The implementation varies considerably. For example, with an operating system, containing full tasking support and memory protection, this includes a switch into kernel mode which is often implemented utilising trap instructions. On the other hand, with real-time operating systems with thread implementation, this is mostly reduced to a simple subroutine call to the operating system kernel.

The measurement must differentiate between three kinds of system calls:

- blocking
- releasing
- local

A blocking system call may not return immediately due to inaccessible data. Typical examples for such blocking calls are semaphore obtain, receive or wait calls. If these calls are blocked, the calling thread is no longer ready to be executed and, therefore, a rescheduling is initiated. With all, but the semaphore obtain, it is assumed that the blocking system call is a method to control the time of thread release. In this way, no further impact on the blocking time B_i is incurred. In contrast, *releasing system calls* always include a rescheduling, since a thread with higher priority might become ready for execution due to the execution of the call. Common examples of such system calls would be a semaphore release or a send call. System calls which do not include a call to the scheduler are considered to be *local*, since the execution of these system calls may not cause a preemption of the calling thread. In this category fall, for example, calls to device drivers.

A basic decision must be made whether to guard system calls with measurement points, or to measure the case where no rescheduling is necessary while adding a preemption penalty for potential rescheduling. Due to the central role in the operating system an instrumentation to enforce paths is all but impossible for the measurement of the applications. Thus, it is better to guard the system calls with additional measurement points. With blocking and releasing system calls two measurement points are usually used. All measurement points are placed as near as possible to that part of the system call containing the scheduling call. Depending on the structure of a local system call, and the reliability requirements, it might be possible to only have to place a single, or indeed no additional measurement points. This is due to the fact that system calls in real-time operating systems are comparably small and simply structured. System calls which correspond to the given properties may be subject to that simplification. In order to be able to omit the measurement points entirely, it is necessary that the utilisation of caches and branch prediction only differ insignificantly, or not at all. System calls which have a slightly more diverging use of caches and branch prediction need a single measurement point at the end of the system call. This may be done when the utilisation of the caches by the system call is variable in location, but fixed in size. In this way, the additional measurement point ensures the worst case state at the beginning of the user code, and the deviation of the system call execution time adds to the statistic properties of the worst case execution time of the thread.

4.5.3 Preemption

The first step must consider the sources of preemption.

- blocking and releasing system calls
- a thread with higher priority becomes ready
- interrupts

With blocking and releasing system calls, the preemption penalty is already considered as described in Section 4.5.2. The transition from *blocked* to *ready* state for a thread is only possible due to the release of a blocking system call, or at the initial startup of this thread. The unblocking of a system call by another thread has already been considered. The unblocking by an interrupt is the only alternative which needs to be taken into account. The interrupts may be differentiated in simple and complex interrupts. Corresponding to the description in Section 4.3.7, simple interrupts may be regarded as an additional statistical factor on the execution time of the thread, on condition that the impact of the interrupt service routine may be considered small on the execution time of the thread, and the interrupt load can be reproduced during measurement. Interrupts which unblock a thread may not be considered in such a way.

The preemption leads to a extrinsic, i. e.; inter-thread modification of the working set of a thread within the acceleration techniques of the processor. The possibilities for covering the extrinsic impact of a preemption on the cache have been classified in [16].

1. The time needed to refill the entire cache.
2. The time needed to refill the cachelines displaced by the preempting thread.
3. The time needed to refill the cachelines used by the preempted thread.
4. The time needed to refill the maximum number of useful cachelines that the preempted thread may hold in cache at the worst case instant a preemption may arise. Useful lines are those that are potentially used again by this thread (similiar to those referred in [55]).
5. The time needed to refill the intersection of cachelines of the preempting and useful cachelines of the preempted thread as described by Lee et al. in [51].

With modern architectures, the first approach leads to severe overestimations due to the large caches. The number of cachelines used by a thread usually exceeds by far the number of usefull cachelines of this thread, at any point during its execution. Therefore approaches 2 and 3 are also very pessimistic. The final method provides the closest results but is almost impossible to determine given the complexity of threads running on modern architectures.

Apart from the caching effects, the branch prediction and other acceleration techniques must be taken into account. Depending on the processor, the possibilities of handling this for the preemption delay differ and must be chosen according to the processor under investigation. As indicated in Sections 4.3.7 and 4.4.7, the impact of preemption on the acceleration techniques can be determined by utilising measurements. The extrinsic interference is covered, taking into account the loss of “good” information within the acceleration units working set of the preempted thread, and not the worst case impact imposed by the preempting thread. A few examples illustrate this. With regard to the caches, the worst case loss of useful cachelines in the preempted thread is considered as opposed to those cachelines loaded by the preempted thread. This corresponds to approach 4 in the classification given previously. TLB entries can be handled accordingly. To provide the impact of the preempting thread on the branch prediction, independent of the preempted thread, is rather challenging. Here as well, the loss of prediction accuracy regardless of the preempting thread is taken into account.

4.5.4 Thread Model

Various information is needed for the real-time analysis described in Section 4.7. It is assumed that after an bootup phase, a thread only has three states, i. e.; *ready*, *running* and *blocked*, and that no two threads share the same priority.

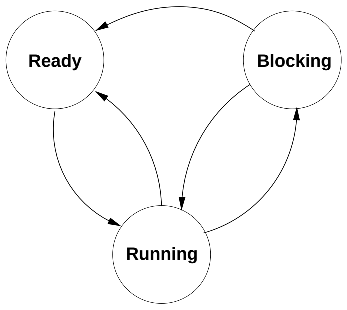


Figure 4.13: State Space of a Thread

A thread in the blocking state is not ready for execution, due to the blocking caused by a system call. This is also known as the inactive state of a thread. After the blocking state, the thread becomes either running or ready to be executed. In the ready state, the thread is not executed due to a running of a thread with higher priority, or the execution of operating system code. As depicted in Figure 4.13, a thread may not change from ready state into blocked state in RT-Linux, since a blocking system call must be executed to initiate the blocking state.

As shown in Section 4.5.1.3, there can be two reasons for a blocking state. Either the blocking is caused by concurrent access to resources guarded by a semaphore, or it is used to control the execution of a process. The most simple example of such a call is the `pthread_wait()` call. This is used to let the thread *sleep* until the next, usually periodic, execution is necessary. Another example, which is not implemented in RT-Linux, is a blocking receive call. This is often used for message passing controlled execution in server threads. Such threads provide a frequently used functionality, and are activated due to the receival of new data. A typical application area for such a server thread would be to circumvent the mutual exclusion problems by semaphores. These threads are only executed after a complete message is received, and ensure the consistant handling of the I/O to peripheral units.

Chains of such threads, activating one after another and utilising message passing, or wake up calls, are concatenated for the real-time analysis. To apply this only threads belonging to this chain may have a priority within the priorities used by the thread chain. Since interrupts triggering a specific thread do not comply with this requirement, they may not be included in the activity chain.

Server threads used by more than one thread are regarded in the same way as semaphore guarded critical section, with ceiling semaphore protocol priority avoidance protocol. A necessary restriction for this to be suitable is that the client thread with the highest priority using a server thread must have the priority just below the server thread.

To facilitate further discussions activity chains of threads are handled as a single thread with a priority with the lowest priority within the chain. The WCET of the participating threads are joined, as described given in Section 4.6.3, and server threads are regarded as semaphore guarded critical sections. The following information is necessary for every thread and interrupt .

- thread or interrupt priority
- starting point and end point of critical sections guarded by a semaphore
- WCET of the thread C
- WCET of critical sections guarded with a semaphore
- triggering interrupt stream corresponding to Section 4.1

The information on the semaphore guarded critical sections is necessary to compute the blocking time B_i. The remaining values are needed for the real-time analysis in Section 4.7.

4.6 Confidence in Measured Values

As previously described in the processor models, in Sections 4.3.6 and 4.4.6, the execution time cannot be fixed entirely, due to the fact that some mechanisms can not be set up in a deterministic way. In order to cope with this nondeterminism a statistical approach aimed at providing a confidence value for the measured WCET (t_{MWCET}), which includes the additional penalties previously mentioned, is presented. The target is to provide an assumed WCET value t_{AWCET} for a given probability p_{AWCET}, so that t_{WCET}, which is the physical WCET, is less or equal to provided value of t_{AWCET}.

The following Section will give an overview on the statistical modelling used.

4.6.1 Statistical Processes

The first step must be to consider the underlying statistical processes. Most of the acceleration techniques cannot be pinned down as having two execution times (one for the good and one for the bad case). Additionally, all acceleration techniques closely interact with the out-of-order execution of the code. The behaviour of the processor for a given piece of code may therefore seem chaotic, i. e.; a small change in one piece of the statistical process may lead to vast changes in the execution time.

A little order is induced by the fact that the processor must wait for external resources to provide the data and code to execute. This waiting cycle, initiated by a stall of the flow of code and data, decouples the intricate interlocking of the sources of statistical effects into a series of statistical processes with discrete probability distributions. Since the caches are filled with useless code and data, a considerable amount of stalls in the execution can be assumed. In general, the process can be considered to be independent, with only a few of the statistical processes being coupled. An example of a coupling mechanism would be the displacement of a cacheline, due to a misprediction in a previous part of the software, which might influence the behaviour of another part of the measurement block. The number of processes could be estimated by utilising the processor PMCs assuming that the processor has to wait at each frontside bus for access to external data and code. The probability distributions for these statistical processes are unknown.

The frontside bus, which is usually clocked with lesser frequency than the processor, leads to a quantisation of the impact from the acceleration techniques, due to the fact that the requested data can only be delivered with one of the cycles of the frontside bus. The measurements support this by showing major peaks as offset at multiples of the frontside bus frequency, and minor values in between. Figure 4.14 prodvides a detailed clip of the sample thread used in Figure 4.15. It has to be noted, that the *y* axis shows the absolute number of measurements with a specific execution time in logarithmic scale. The CPU to FSB frequency ratio was 8 i. e.; the peaks are usually found at multiples of 8 cycles.

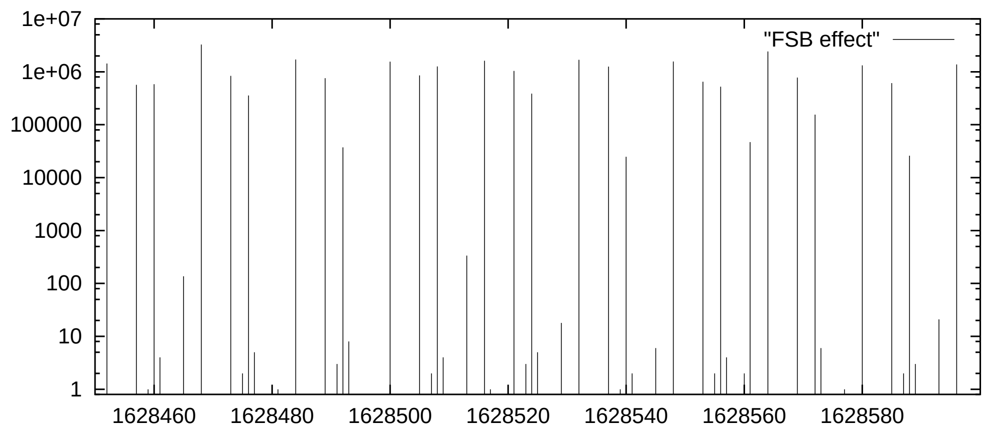


Figure 4.14: Effect of the Frontside Bus on the Execution Time

4.6.2 Probability Distribution of Measurement Blocks

In order to computate a confidence value, the execution time of a measurement block needs to be modelled. While traditional statistic approaches focus on the data close to the mean, the main objective to modelling here are the extreme values. To achieve our objective the *extreme value (EV) statistics* approach of Burns and Edgar in [13] and [14] is used. An essential precondition for the approach is that the population samples, drawn by measurements, $X_1, X_2, \dots X_n$ are independent, and identically distributed (I. I. D.), in other words - all samples have the same underlying *probability density function (pdf).* To have the right starting conditions this means that the individual condition must either be identical, or completely random for all measurements. This is ensured by randomising those parts of the processor which can not be set up deterministically, to the worst case state at the start of a measurement.

Similar to the normal distribution the distributions modelling accurately the minimum and maximum of a given sample distribution, are chosen in such a way that they possess important properties which allow the modelling of distributions generally. The extreme value distributions having these properties have the following definitions for their *cumulative distribution function (cdf)*:

$$\begin{array}{llcll}
\text{Gumbel} & G_0(x) & := & \exp\left(-e^{-x}\right) & \\
\text{Frèchet} & G_{1,\alpha}(x) & := & \exp\left(-x^{-\alpha}\right), & \alpha > 0 \\
\text{Weibull} & G_{2,\alpha}(x) & := & \exp\left(-\left(-x\right)^{-\alpha}\right), & \alpha > 0
\end{array}$$

The term $\exp\left(-e^{\cdots}\right)$ is usually used to describe a double exponential function in extreme value statistics. It is equivalent to $e^{-e^{\cdots}}$. A common notation for all three distributions is:

$$G_\gamma(x) := \exp\left(-(1+\gamma x)^{-\frac{1}{\gamma}}\right), \quad 1+\gamma > 0 \tag{4.9}$$

This cdf suffice for the Fisher–Tippett theorem:

Theorem 4.1:[Fisher–Tippett] *If $F(ax+b)$ has a non-degenerate limiting distribution function for constants $a > 0$ and b as $n \to \infty$ then:*

$$\left| F^n(x) - G\left(\left(\frac{x-\mu_n}{\sigma_n}\right)\right)\right| \to 0 \quad as \quad n \to \infty \tag{4.10}$$

for an EV function G and μ_n and $\sigma_n > 0$ drawn from the sample distribution [26].

Since the parameters σ and μ are computed from the samples taken, only the shape parameter α or γ, depending on the model chosen, needs to be identified. This parameter can be calculated using a *maximum likelihood estimator*. Since the Gumbel distribution provides good matches on the data measured, and the matching of parameters is out of scope for this work, further discussion will be limited to the Gumbel distribution with cdf:

$$G(t) \quad = \quad \exp\left(-e^{\frac{t-\mu}{\sigma}}\right) \tag{4.11}$$

and pdf:

$$g(t) \quad = \quad \exp\left(-e^{-\frac{t-\mu}{\sigma}}\right) * \frac{e^{-\frac{t-\mu}{\sigma}}}{\sigma} \tag{4.12}$$

Special focus for the discussion is set on the right distribution tail of the distribution. Figure 4.15 shows an example measurement, and the match produced with the Gumbel distribution. Due to the previously described effect of the frontside bus, the measured discrete distribution of the figure is displayed as a *kernel density*, which is a method to transform discrete data into a continuous curve.

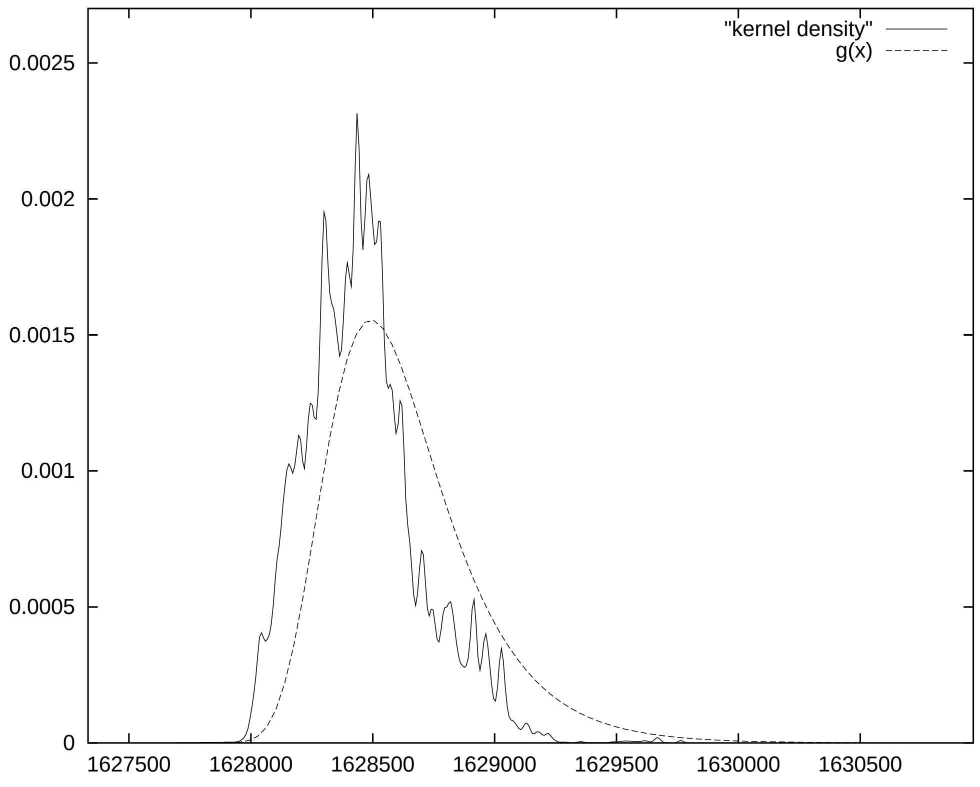


Figure 4.15: Sample Data and Corresponding EV–Distribution

The major drawback of modelling the execution time with a Gumbel probability distribution functions is that the cdf reaches the 100 % certainty only in positive infinity. Thus a guarantee in the form of

There is X % confidence that $t_{AWCET} > t_{WCET}$.

is all but impossible. However, an approach used often in reliability analysis may be applied here. Many components cannot be guaranteed with 100 % certainty not to fail. An typical example is natural aging of components (mechanical or electronic) during the lifetime of a system. Even with continous quality control and preventive maintenance an element of risk of a component failure remains. The residual risk acceptable depends on the consequences of a component failure. A component inhibiting operation of an air conditioning unit in a car wont have the residual risk requirements as another component responsible for the operation of a air traffic controller computer system at a large international airport. The tolerance level for the malfunction of such systems λ is often given in failures during 10^6 hours of operation (cf. [80]). Typical values for λ are 1, 10^{-3}, ... and 10^{-12}. In real-time systems a piece of software may be specified malfunctioning if the execution time takes longer than the assumed WCET t_{AWCET}. Since a thread or task is usually running many times during an hour of operation of the system, the resulting residual risk acceptable for a single run of the thread or task is usually several orders of magnitudes less then the residual risk allowed for the overall system. In accordance with Section 4.1 Equation 4.13 computes the necessary cdf value acceptable in the Gumble distribution.

$$G_i(t) \geq 1 - \frac{\lambda_i}{E_i(10^6 hours)} \tag{4.13}$$

Table 4.1 depicts the necessary distance to the mean value of a gumble distribution to achieve a given confidence in the measured value. While a distance of $55.3 * \sigma$ seems large to achieve the desired confidence. The results in Section 5.4 show, that the resulting distance in processor cycles is usually an order of magnitude or more smaller than the mean value measured. This stems from the fact, that the worst case state is taken as a starting point for the measurements and only a single path is measured.

4.6.3 Joining of Measurement Blocks

The WCET of a thread is comprised by the sum of WCET of measurement blocks. As defined in the previous section, the WCET of a single measurement block is not presented as a single WCET value, but as a probability density function of the execution time.

The combination of the measurement blocks has to be considered carefully. In this discussion two probability density functions $g_1(t)$ and $g_2(t)$, based on the Gumbel distri-

$G(t)$	t
$1-10^{-6}$	$13.8*\sigma+\mu$
$1-10^{-9}$	$20.7*\sigma+\mu$
$1-10^{-12}$	$27.6*\sigma+\mu$
$1-10^{-15}$	$34.5*\sigma+\mu$
$1-10^{-18}$	$41.4*\sigma+\mu$
$1-10^{-21}$	$48.3*\sigma+\mu$
$1-10^{-24}$	$55.3*\sigma+\mu$

Table 4.1: Neccessary Distance to Mean to Achieve a Given Confidence

bution, are taken into account for the two blocks under investigation. According to the previous Section these functions are characterised by their means (μ_1 and μ_2) and deviations (σ_1 and σ_2).

In general, both cases of alternative and consecutive blocks have to be considered. Consecutive blocks may be joined by convoluting the density functions of both blocks. The proof of this operation can be found in the convolution theorem 2.7.5 in [2]. A base requirement for the validity of the operation, is the independence of the two statistic processes, i. e.; the measured execution time of one block may not influence the measured time of the other. This is guaranteed by the way the measurement routine handles the processor's acceleration techniques.

$$g_r(t) = \int_{-\infty}^{+\infty} g_1(x)g_2(t-x)dx \tag{4.14}$$

or fully

$$g_r(t) = \int_{-\infty}^{+\infty} \frac{\exp\left(-e^{-\frac{x-\mu_1}{\sigma_1}}\right)*e^{-\frac{x-\mu_1}{\sigma_1}}}{\sigma_1} * \frac{\exp\left(-e^{-\frac{t-x-\mu_2}{\sigma_2}}\right)*e^{-\frac{t-x\mu_2}{\sigma_2}}}{\sigma_2} dx \tag{4.15}$$

The solution of Equation 4.15 is not trivial, and currently no analytical solution is known. Fortunately, a numerical solution shows that the convolution of two Gumbel extreme value probability density functions results in something close to a Gumbel pdf. Figure 4.16 shows the numerical convolution of two Gumbel pdfs with identical sigmas, and a simple aproximation computed with the following parameters as derived from the convolution of arbitrary independent distributions.

$$\sigma_r = \sqrt{\sigma_1^2+\sigma_2^2} \tag{4.16}$$

$$\mu_r = \mu_1+\mu_2 \tag{4.17}$$

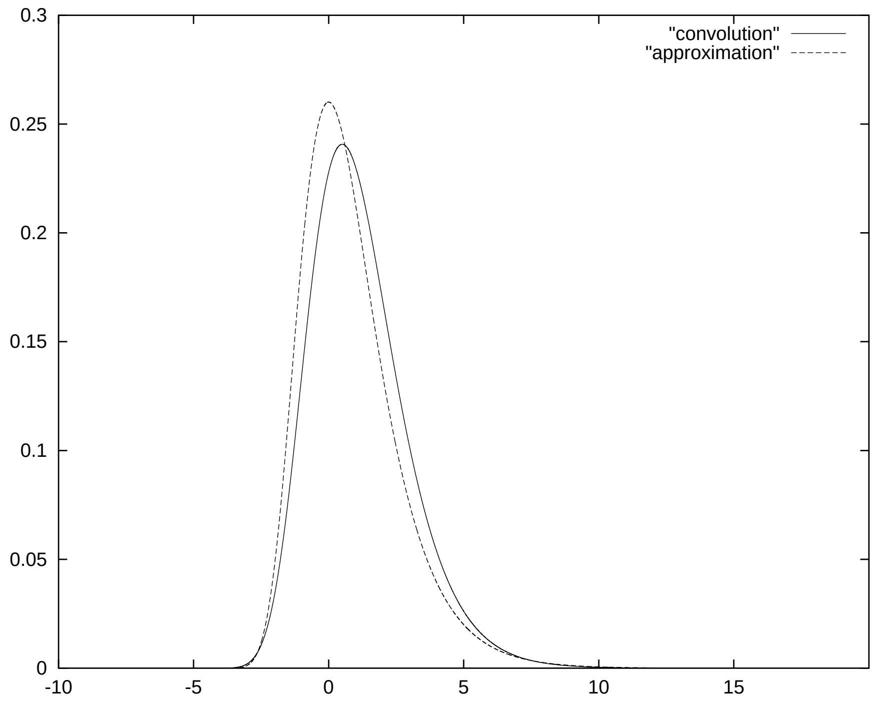


Figure 4.16: Convolution Result and Simple Approximation

Identical sigmas have been chosen, since these produce the largest difference. It has to be noted that the convolution of many individual distributions as we have in our case tends towards a resulting Gaussian distribution which describes better the average case then the extremes. In order to retain this modelling of the extremes an approximation with a Gumbel distribution is done. The approximation as shown in Figure 4.16 is not good enough for practical purposes, since the pdfs are underestimated. The further approximation was done in three steps:

1. Search for the mean of the new Gumbel pdf by searching the value e^{-1} in the cdf of the convoluted function.

$$\mu = p^{-1}_{cdf,convoluted}(e^{-1}) \tag{4.18}$$

2. Find the sigma that minimises the quadratic error of the cumulative distribution function between the matched Gumbel distribution and the convoluted distribution function.

3. Move the matched cdf so that the following equation holds true:

$$\forall t > \mu : \quad p_{cdf,convoluted}(t) \geq p_{cdf,matched}(t) \tag{4.19}$$

The first step determines the invariant point for the second step of the approximation. The final step is necessary to ensure that the approximation gives an upper bound on the probability, for a given execution time. A variety of combinations of σ_1 and σ_2 have been tested to determine the dependency of the offsets, and standard deviation.

Figure 4.17 shows parameter y of the standard deviation. The final standard deviation is computed using the following formula.

$$\sigma_j = (1+y) * \sqrt{\sigma_1^2 + \sigma_2^2} \tag{4.20}$$

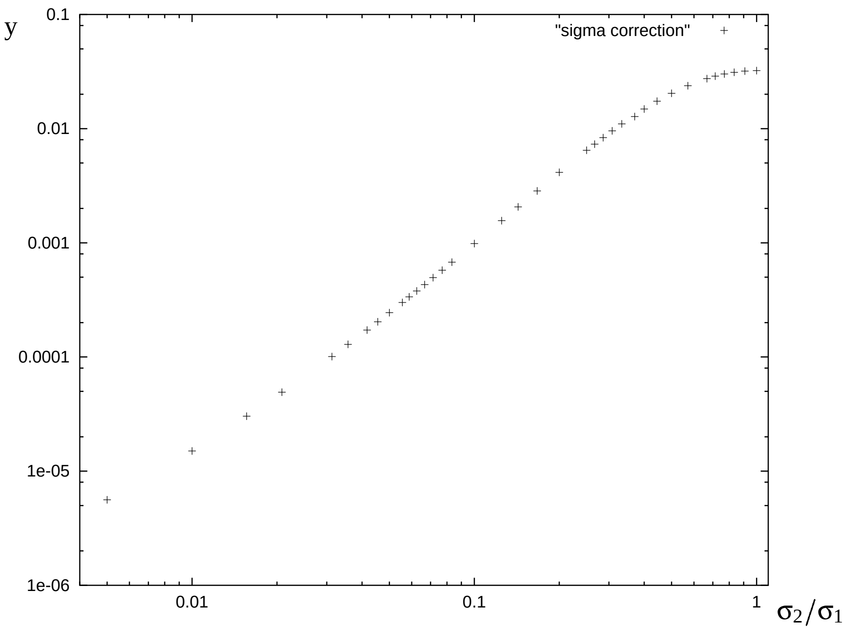


Figure 4.17: Correction Factor for σ-Computation

The resulting offset relative to μ_r is depicted in Figure 4.18. In this way, the offset μ_j can be computed using Equation 4.25.

The offset and sigma can be calculated for the computation, using the following Table 4.2. It has to be noted that, for this calculation, σ_1 has been chosen to be greater or equal to σ_2. With the other case, the parameters have to be swapped correspondingly.

The additionally inserted approximation in Figure 4.18 shows an interesting behaviour.

$$z_{approximation} = \frac{x}{1.75} * \sigma_1 \tag{4.21}$$

$$= \frac{\sigma_2}{\sigma_1 * 1.75} * \sigma_1 \tag{4.22}$$

$$= \frac{\sigma_2}{1.75} \tag{4.23}$$

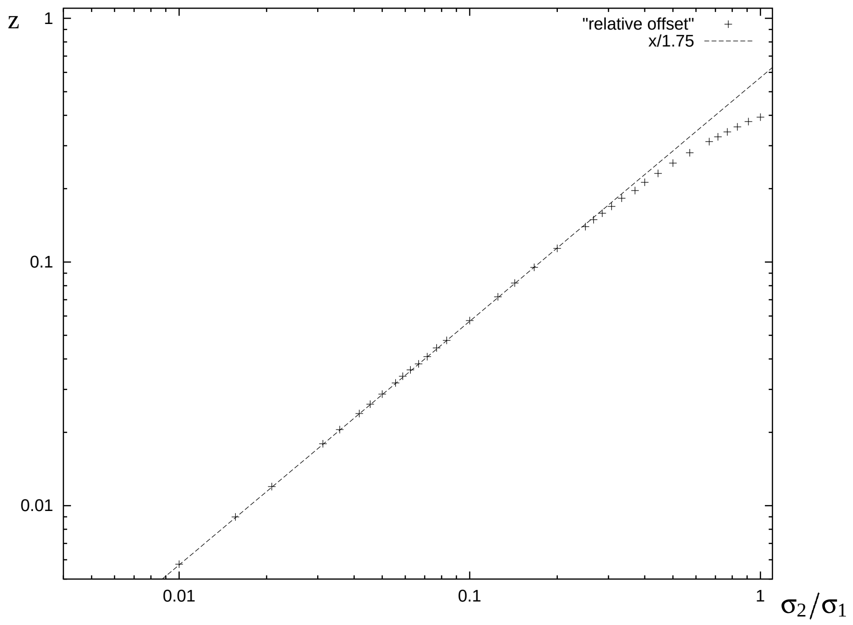


Figure 4.18: Correction Factor for Offset Computation

Thus the offset can be expressed as a simple function of σ_2. The aproximation is reasonable for $10 * \sigma_2 < \sigma_1$. If σ_2 is larger, then this approximation degrades and a interpolation using the data in Table 4.2 yields better results.

In the case of alternative blocks, two scenarios have to be considered. It is assumed that probability functions A and B are described by μ_A, μ_B, σ_A, and σ_B with $\mu_A > \mu_B$. The first case is $\sigma_A < \sigma_B$. Let us assume that a few convolutions have already been approximated in parallel, i. e.; without deciding which of the alternatives is taken. Under these conditions it can be assumed that the deviations describing the Gumbel distributions are larger than all other standard deviations to be merged in the future process. In this way, the move of the offset μ_j will always be larger for the function A. Thus an alternative can be neglected if its sigma holds true for the following equation:

$$\sigma_A \quad > \quad \sigma_B * (1+y)^N \tag{4.26}$$

N is the number of blocks to be merged to finalise the join process, and y is chosen from Table 4.2, according to the function of the largest σ to be joined in. Equation 4.26 bounds the gain in the deviation. The actual gain will be less, since the closer the two alternative sigmas become, the less the gain actually taken when compared to function A.

The second case of $\sigma_B > \sigma_A$ can not be decided on a general basis. Thus both alternatives have to be pursued until either μ_B surpasses μ_A and Equation 4.26 holds true, or all blocks have been merged and the larger value is chosen for the WCET. A simplistic method to work around the problem of having to account for exploding number of paths is to choose

$x = \sigma_2/\sigma 1$	y	z
1	$3.225e-2$	$0.3930*\sigma_1$
0.909	$3.195e-2$	$0.3770*\sigma_1$
0.833	$3.118e-2$	$0.3587*\sigma_1$
0.769	$3.009e-2$	$0.3418*\sigma_1$
0.714	$2.881e-2$	$0.3263*\sigma_1$
0.667	$2.740e-2$	$0.3120*\sigma_1$
0.571	$2.379e-2$	$0.2808*\sigma_1$
0.5	$2.039e-2$	$0.2547*\sigma_1$
0.444	$1.741e-2$	$0.2310*\sigma_1$
0.4	$1.488e-2$	$0.2124*\sigma_1$
0.370	$1.277e-2$	$0.1965*\sigma_1$
0.333	$1.102e-2$	$0.1826*\sigma_1$
0.308	$9.558e-3$	$0.1692*\sigma_1$
0.286	$8.341e-3$	$0.1586*\sigma_1$
0.267	$7.322e-3$	$0.1492*\sigma_1$
0.25	$6.465e-3$	$0.1397*\sigma_1$
0.2	$4.142e-3$	$0.1138*\sigma_1$
0.167	$2.847e-3$	$0.0951*\sigma_1$
0.143	$2.066e-3$	$0.0820*\sigma_1$
0.125	$1.564e-3$	$0.0720*\sigma_1$
0.1	$9.838e-4$	$0.0575*\sigma_1$
0.0833	$6.765e-4$	$0.0477*\sigma_1$
0.0769	$5.749e-4$	$0.0444*\sigma_1$
0.0714	$4.950e-4$	$0.0409*\sigma_1$
0.0667	$4.310e-4$	$0.0382*\sigma_1$
0.0625	$3.789e-4$	$0.0361*\sigma_1$
0.0588	$3.360e-4$	$0.0340*\sigma_1$
0.0556	$3.003e-4$	$0.0319*\sigma_1$
0.05	$2.445e-4$	$0.0287*\sigma_1$
0.0455	$2.034e-4$	$0.0261*\sigma_1$
0.0417	$1.723e-4$	$0.0239*\sigma_1$
0.0357	$1.290e-4$	$0.0205*\sigma_1$
0.0313	$1.008e-4$	$0.01796*\sigma_1$
0.0208	$4.915e-5$	$0.01198*\sigma_1$
0.0156	$3.033e-5$	$0.00899*\sigma_1$
0.01	$1.502e-5$	$0.00575*\sigma_1$

$$\sigma_j = \sqrt{\sigma_1^2+\sigma_2^2}*(1+y) \tag{4.24}$$

$$\mu_j = \mu_1+\mu_2+z \tag{4.25}$$

Table 4.2: Parameters for the Approximation of the Convolution

the largest μ and σ observed in the measurements of the different paths of a particular block. Thus the worst case is still bound, but obviously this is bought with additional pessimism.

4.7 Real-time Analysis

As a real-time analysis method *response time analysis (RTA)* is chosen. The analysis method presented in [12] builds the basis of the RTA presented here. Sufficient and necessary tests must be performed for an exact schedulability analysis. This is often done by simulating the schedule over an interval of the least common multiple of thread periods. In the presence of sporadic threads, i. e. , threads having an inter-arrival time of releases which are only bound to be larger than, or equal to, the given minimum interval, this is not possible. As an alternative for the interval-based approach, it is tested how often a given thread τ_j might be released, in the worst case in the execution window R_i, in which thread τ_i is executed. Due to preemption and blocking, the execution window *is not* the same as the worst case execution time. Thus a phasing of the release points of a thread is not necessary.

In contrast to the approach in [12] the following assumptions are made:

- According to Section 4.5.3 a preemption of thread τ_i incurs a non-zero individual time penalty δ_i suffered by the thread. This penalty δ_i includes, not only the time needed for the scheduling algorithm, but also extrinsic interferences like, for example, displacement of cachelines by the preempting task, which might be useful to the tasks future execution, or for perturbation of the branch prediction mechanism. As previously mentioned, due to the nature of the worst case view on the measurement blocks (described in greater detail in Section 4.5.4), it is more favourable to determine the preemption penalty by the thread preempted, and not by the preempting thread as is done by Busquetes et al. in, for example, [16] (cf. Section 2.4).

- The task releases are determined by interrupt streams IS, in accordance with Section 4.1. Therefore, the maximum possible number of releases of thread τ_i, in interval T, is $E_i(T)$.

4.7.1 Response Time Analysis of a Simple System

Starting point is an RTA approach, which calculates the worst case response time R_i for each thread τ_i. The worst case response time is produced, when all threads are released at the same point in time. The recursive approach tries to allocate the computational time C_i, and blocking time B_i of thread τ_i, in a time window w_i, and the simple interference of all threads of higher priority than τ_i. The priority of thread τ_i is fixed and unique, with the exception of the priority inheritance protocol, as described in Section 4.5.1.3. For this simple approach to be applicable the following restrictions are necessary:

1. The release time of thread τ_i is characterised by the minimum inter-arrival time T_i.
2. Preemption incurs no time penalty on preempted threads.

These restrictions will be lifted in the next section as the approach is fitted to the system described so far i. e. with preemption penalties, interrupt streams triggering the execution of threads and priority inheritance protocol.

The approach is recursive since in every iteration the execution window w_i^n has to be checked whether additional interference with other threads has to be added. The process finishes either when the execution window stops growing i. e. $w_i^{n+1} = w_i^n$ and thus the worst case response time R_i has been found or when the execution window exceeds the deadline of the thread and thus the deadline is potentially missed i. e. $w_i^{n+1} > D_i$. The latter case indicates that the thread cannot be guaranteed to execute without violating its deadline. For simplicity reasons, it is assumed that the threads are sorted in priority order with τ_1 being the highest priority thread.

$$w_i^{n+1} = C_i + B_i + \sum_{\tau_j \in \mathbf{H}_i} \underbrace{\left\lceil \frac{w_i^n}{T_j} \right\rceil}_{N_j} * C_j \tag{4.27}$$

Initially the execution window w_i^1 is assumed to be the computation time C_i of thread τ_i, though one can find other more efficient starting values for w_i^1. Since all threads are released at the same time, all threads with higher priority than τ_i which form the set $\mathbf{H}_i$, are released at least once during w_i^n and their computation times C_j have to be added according to the number of releases which is tagged with N_j in Equation 4.27.

4.7.2 Extending the Simple Real-Time Analysis for Complex Systems

As previously mentioned, the complexity of the system under investigation does not allow for this simple solution, and therefore the Equation 4.27 must be extended considerably.

To facilitate the theorem as regards the response time analysis and the corresponding proof, a few terms, not yet defined in this work need to be introduced:

$\Delta_{i,j}(R_i)$ is the worst case additional preemption delay suffered by thread τ_i, due to releases of thread τ_j in the interval R_i.

$P^n_{i,j}$ is the worst case number of preemptions of thread τ_i by thread τ_j, which are not covered at start of iteration n.

$\mathbf{S}^n_{i,j}$ is the set of threads which potentially suffer preemption by τ_j, instead of τ_i, and have not been covered at start of iteration n.

$\Theta^n_{i,j}$ The cumulative preemption delay caused by thread τ_j, on thread τ_i, after $n-1$ iterations.

$\min(\ldots)$ provides the minimum value of it's arguments.

For those systems including the additional extrinsic interference δ_i, and non equidistant thread releases where $E_j(t)$ expresses the worst case number of releases of τ_j in the interval t, the following theorem is stated:

Theorem 4.2: *Given a set of real-time threads scheduled by a fixed priority preemptive policy with priority inheritance for priority inversion protection in a system where thread τ_i suffers a worst case penalty of δ_i for every preemption (all threads are required to comply with the requirements of conventional response time analysis), then there either exists a worst case response time value for R_i for each thread τ_i making the equation system depicted below true, or such a thread is not schedulable.*

$$R_i = C_i + B_i + \sum_{\tau_j \in \mathbf{H}_i} \left(E_j(R_i) * C_j + \Delta_{i,j}(R_i)\right) \tag{4.28}$$

$\Delta_{i,j}(R_i)$ is computed iteratively by the following formula.

Initialisation:

$$\begin{aligned} P^1_{i,j} &= E_j(R_i) \\ \mathbf{S}^1_{i,j} &= \tau_i \cup \mathbf{H}_i \cap \mathbf{L}_j \\ \Theta^1_{i,j} &= 0 \end{aligned} \tag{4.29}$$

Iterative process:

$$\delta_k = \max\left(\delta_l : \tau_l \in \mathbf{S}^n_{i,j}\right)$$

$$\begin{aligned}
\mathbf{S}_{i,j}^{n+1} &= \mathbf{S}_{i,j}^{n} \setminus \tau_k \\
\Theta_{i,j}^{n+1} &= \Theta_{i,j}^{n} + \min\left(P_{i,j}^{n}, E_j(R_k) * E_k(R_i)\right) * \delta_k \\
P_{i,j}^{n+1} &= P_{i,j}^{n} - E_k(R_j) * E_k(R_i)
\end{aligned} \tag{4.30}$$

Terminal condition:

$$P_{i,j}^{n+1} \leq 0 \tag{4.31}$$

After the iterative formula has terminated:

$$\Delta_{i,j}(R_i) = \Theta_{i,j}^{n+1} \tag{4.32}$$

Proof: Due to the dependency on the worst case response times of those threads which have a higher priority than thread τ_i, it is necessary that the real-time analysis is conducted in priority order, starting with the highest priority. To simplify the proof, it is assumed that the worst case response time R_i of thread τ_i has been found, i. e. , $w_i^n = w_i^{n+1}$, thus all occurrences of w_i^n and w_i^{n+1} are replaced with R_i .

As stated in the previous section, the simple RTA formula is taken as a basis.

$$R_i = C_i + B_i + \sum_{\tau_j \in \mathbf{H}_i} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \tag{4.33}$$

In the first instance, the number of releases of thread τ_i must be matched to the model of the embedding process as described in Section 4.1. Since $E_j(R_i)$ returns the maximum number of releases of thread τ_j in interval R_i, $\left\lceil \frac{R_i}{T_j} \right\rceil$ may be simply replaced in Equation 4.33. Thus this equation can be rewritten as:

$$R_i = C_i + B_i + \sum_{\tau_j \in \mathbf{H}_i} E_j(R_i) * C_j \tag{4.34}$$

Thereafter, the additional extrinsic interference must be introduced into Equation 4.34. Assuming the response time of thread τ_2 is investigated, and τ_2 suffers preemption of one thread with higher priority. Whenever τ_2 is preempted by τ_1 after resuming execution, thread τ_2 has to pay the additional preemption delay δ_2. A sample phasing is depicted in Figure 4.19.

Thus, with respect to thread τ_2, the Equation 4.33 would be extended and simplified to:

$$R_2 = C_2 + B_2 + E_1(R_2) * (C_1 + \delta_2) \tag{4.35}$$

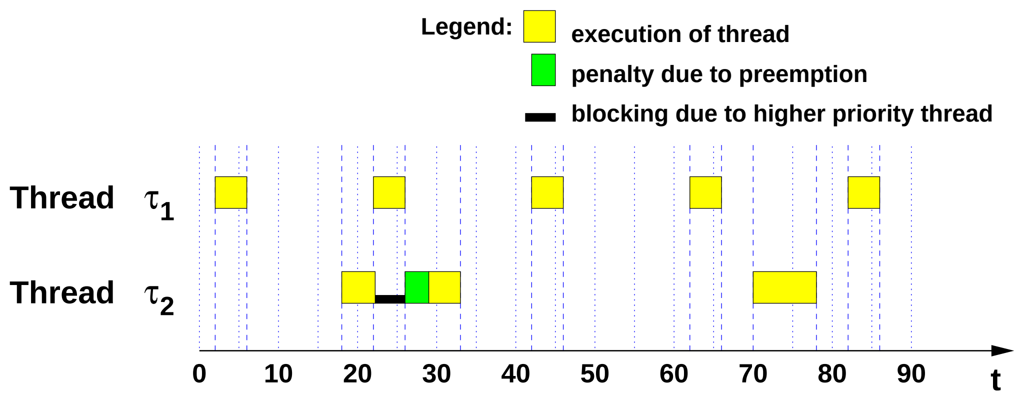


Figure 4.19: Preemption of Thread τ_2 by Thread τ_1

A slightly different picture must be drawn if the preemption delay of thread τ_3 is covered. First of all, the preemption of thread τ_3 by τ_2 is investigated. This is analogous to the preemption of τ_2 by τ_1. But the preemption of thread τ_3 by τ_1 has to handled differently. In this case two scenarios have to be considered, both of which are depicted in Figure 4.20.

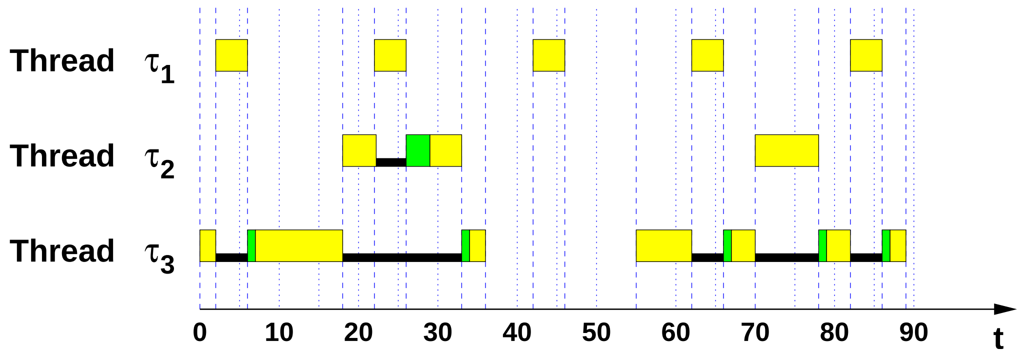


Figure 4.20: Preemption of Thread τ_3 by Threads τ_2 and τ_1

Within the interval defined by the execution window of thread τ_3, thread τ_1 is released twice and thread τ_2 once. The first occurrence of thread τ_1 preempts τ_3, while in the second case τ_2 is preempted. In the second release of thread τ_3, this thread is preempted twice by τ_1 and once by τ_2. In order to account for the worst case preemption delay, the maximum of preemption delays has to be taken into account. Thus the Equation 4.35 needs to be extended by the term $\max(\delta_2, \delta_3)$ for one preemption induced in R_3^n by thread τ_1, and twice δ_3 for one preemption of thread τ_1 and of τ_2 each. In more general terms, the additional preemption delay on thread τ_3 is δ_3 for all preemptions by thread τ_2, and for preemptions by thread τ_1 the delay is the $\max(\delta_2, \delta_3)$ for at most the number of times

τ_1 preempts one execution of τ_2, multiplied the number of times τ_2 preempts τ_3. In Equation 4.36 this is presented in a more formal way.

$$\begin{aligned} R_3 &= C_3 + B_3 + E_2(R_3) * (C_2 + \delta_3) + E_1(R_3) * C_1 + \\ &\quad \max(\delta_2, \delta_3) * E_1(R_2) * E_2(R_3) + \\ &\quad \delta_3 * (E_1(R_3) - E_1(R_2) * E_2(R_3)) \end{aligned} \tag{4.36}$$

It needs no mathematical proof to see that the complexity of this formula would vastly increase if extended to more than three threads. On the other hand, the additional preemption delay to be taken into account for thread follows simple rules which will be explained in detail below. Thus the additional preemption delay is abstracted and replaced by the variable $\Delta_{i,j}(R_i)$, resulting in the following form where $\Delta_{i,j}(R_i)$ represents the additional preemption delay suffered by thread τ_i, due to the preemption by thread τ_j:

$$R_i = C_i + B_i + \sum_{\tau_j \in \mathbf{H}_i} \left(E_j(R_i) * C_j + \Delta_{i,j}(R_i) \right) \tag{4.37}$$

In order to determine $\Delta_{i,j}(R_i)$, let us consider first which possible scenarios need to be dealt with whenever thread τ_j potentially preempts thread τ_i. The release of thread τ_j may only preempt threads of lower priority than thread τ_j. To compute the additional preemption delay of thread τ_i, only threads with higher priority than thread τ_i, and τ_i itself, need to be considered. Thus, the set $\mathbf{S}_{i,j}$ of threads which have to be taken into account for the additional preemption delay for thread τ_i due to preemption by thread τ_j, is comprised of $\tau_i \cup \mathbf{H}_i \cap \mathbf{L}_j$.

Out of set $\mathbf{S}_{i,j}$, the thread τ_k which suffers the largest individual additional preemption delay δ_k whenever it is preempted has to be chosen. Next, one has to consider how often thread τ_k may be preempted by thread τ_j, in its execution window R_k. This has been solved before in Equation 4.2. In this way the number of preemptions suffered in the worst case by thread τ_k, during its response time R_k by thread τ_j, is $E_j(R_k)$. Combined with the worst case number of preemptions thread τ_i suffers by τ_k, during one execution of τ_i which is $E_k(R_i)$, it can be computed how many times thread τ_k might suffer preemption instead of thread τ_i. Thus the additional preemption delay $\theta_{i,j,k}$ can be computed by Equation 4.38.

$$\theta_{i,j,k} = E_j(R_k) * E_k(R_i) * \delta_k \tag{4.38}$$

In the next iteration the remaining preemptions of $E_j(R_i)$, not covered by Equation 4.38, must be considered. The number of preemptions not already considered are present in $P^n_{i,j}$. The second worst case thread is chosen out of $\mathbf{S}_{i,j}$ for further computation. To keep the formula generic, the next worst case thread is chosen out of the set $\mathbf{S}_{i,j} \setminus \tau_k$. In order to avoid having to account for more than $P^1_{i,j}$ preemptions by thread τ_j, the number of preemptions covered in iteration p may not exceed $P^n_{i,j}$. Thus $\min\left(P^n_{i,j}, E_j(R_k) * E_k(R_i)\right)$ are covered within iteration p.

The process is repeated until all preemptions by thread τ_j are covered. In order to represent the limitations outlined above, Equation 4.38 is modified to:

$$\theta_{i,j,k} = \min\left(P^n_{i,j}, E_j(R_k) * E_k(R_i)\right) * \delta_k \tag{4.39}$$

To simplify the iterative process, and avoid the final summation of all $\theta_{i,j,k}$ the Equation can be transformed to:

$$\Theta^{n+1}_{i,j} = \Theta^n_{i,j} + \min\left(P^n_{i,j}, E_j(R_k) * E_k(R_i)\right) \delta_k \tag{4.40}$$

The steps presented in Equations 4.37 to 4.40 can be summarised in the following equations, which correspond to Equations 4.28 to 4.32:

$$R_i = C_i + B_i + \sum_{\tau_k \in \mathbf{H}_i} \left(E_k(R_i) * C_j + \Delta_{i,j}(R_i)\right) \tag{4.41}$$

With $\Delta_{i,j}(R_i)$ computed iteratively:

Initialisation:
The number of preemptions which need to be covered by the analysis are computed by Equation 4.42. The starting set of threads to be considered for the preemption of thread τ_i by thread τ_k, is depicted in Equation 4.43. Equation 4.44 resets the initial value of the preemption delay to zero.

$$P^1_{i,j} = E_j(R_i) \tag{4.42}$$

$$\mathbf{S}^1_{i,j} = \tau_i \cup \mathbf{H}_i \cap \mathbf{L}_j \tag{4.43}$$

$$\Theta^1_{i,j} = 0 \tag{4.44}$$

Iterative process:
Equation 4.45 displays the thread selection, which comprises the worst case individual preemption delay in set $\mathbf{S}^n_{i,j}$. The set is then reduced by the found thread τ_k, in Equation 4.46. The preemption penalty covered by thread τ_k is computed in Equation 4.47,

while in Equation 4.48 the preemptions not covered after this step are computed.

$$\begin{aligned}
\delta_k &= \max\left(\delta_l : \tau_l \in \mathbf{S}^n_{i,j}\right) && (4.45)\\
\mathbf{S}^{n+1}_{i,j} &= \mathbf{S}^n_{i,j} \setminus \tau_k && (4.46)\\
\Theta^{n+1}_{i,j} &= \Theta^n_{i,j} + \min\left(P^n_{i,j}, E_j(R_k) * E_k(R_i)\right)\delta_k && (4.47)\\
P^{n+1}_{i,j} &= P^n_{i,j} - E_j(R_k) * E_k(R_i) && (4.48)
\end{aligned}$$

Terminal condition:
The iteration is finished when all preemptions have been covered. This is presented in Equation 4.49.

$$P^{n+1}_{i,j} \leq 0 \qquad (4.49)$$

Due to the step from Equation 4.39 to Equation 4.40, $\Theta^{n+1}_{i,j}$ holds the value $\Delta_{i,j}(R_i)$ after the last iteration, which can than be put into Equation 4.37.

$$\Delta_{i,j}(R_i) := \Theta^{n+1}_{i,j} \qquad (4.50)$$

The termination of the iteration is ensured, since the set $\mathbf{S}^1_{i,j}$ has a limited countable number of elements, and τ_i leads Equation 4.48 to be:

$$P^{n+1}_{i,j} = P^n_{i,j} - E_j(R_i) * E_i(R_i) \qquad (4.51)$$

where $E_i(R_i) \geq 1$ (cf. Equation 4.2) and $P^n_{i,j} \leq P^1_{i,j} = E_j(R_i)$ (cf. Equation 4.42). Assuming the minimum value for $E_i(R_i)$, and the maximum value for $P^1_{i,j}$, when thread τ_i is chosen in Equation 4.45, Equation 4.48 can be transformed to:

$$\begin{aligned}
& P^{n+1}_{i,j} \leq E_j(R_i) - E_j(R_i) && (4.52)\\
\rightsquigarrow \quad & P^{n+1}_{i,j} \leq 0 && (4.53)
\end{aligned}$$

Equation 4.53 corresponds to the terminal condition in Equation 4.49. In this way, the termination of the iterative process is proven.

5 Experimental Validation

5.1 Methodology Overview

The implementation of the approach can be divided into two parts: The development tool chain for the application, and the analysing/controlling tool. The first part can be used to implement the entire method by hand. While this works well for small scale applications, complex applications take considerable effort. Figure 5.1 gives an overview on the necessary operations for the generation of the measurement, and final executable.

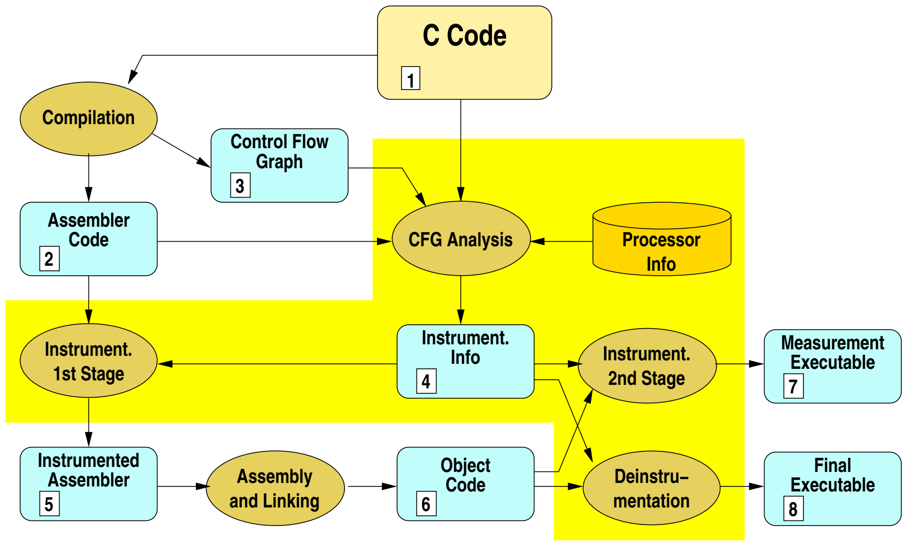


Figure 5.1: Overview of the Operations on the Code

The part highlighted in Figure 5.1 is integrated in the *Path ANalysing tool* $\mathcal{PAN}$, while the compiler, assembler, linker and make program are stand alone tools which are activated by $\mathcal{PAN}$. The second stage of the instrumentation, and the resulting measurement, is an iterative process which must be repeated until all path combinations have been sufficiently measured.

In the following analysis phase of the measurement results the mean and standard deviation of the execution time of each measurement block is computed. Necessary penalties, as described in the processor analysis, are added to the mean. Finally, the execution time distribution function, for the whole thread in accordance with Section 4.6.3 is computed. This analysis phase is not yet integrated into $\mathcal{PAN}$, but consists of a number of small tools supporting a manual analysis.

5.2 Tool Description

As part of the test setup, the following sections will give an overview of the framework built by the stand alone development tools and $\mathcal{PAN}$. Figure 5.2 provides an overview of the structure of the tool $\mathcal{PAN}$ and its interconnectivity with the other stand alone tools used.

5.2.1 Assumptions

Restrictions must be placed on the code in order for the measurement based approach to work.

1. no path independent random access (e. g. lookup-tables)
2. fixed memory locations

Since the processor is assumed to have caches it is important to the reproducibility of the results that the accesses to data are deterministic.

To keep the complexity of the analysing tool to a minimum, a few restrictions and requirements with regards to the code under investigation, have been imposed.

1. The code is limited within one file.
2. No recursion.
3. No indirect function calls.
4. No multiple loops within one line.
5. The number of loop iterations is given in annotations.
6. Annotations indicate whether the paths inside the loop are data dependent.

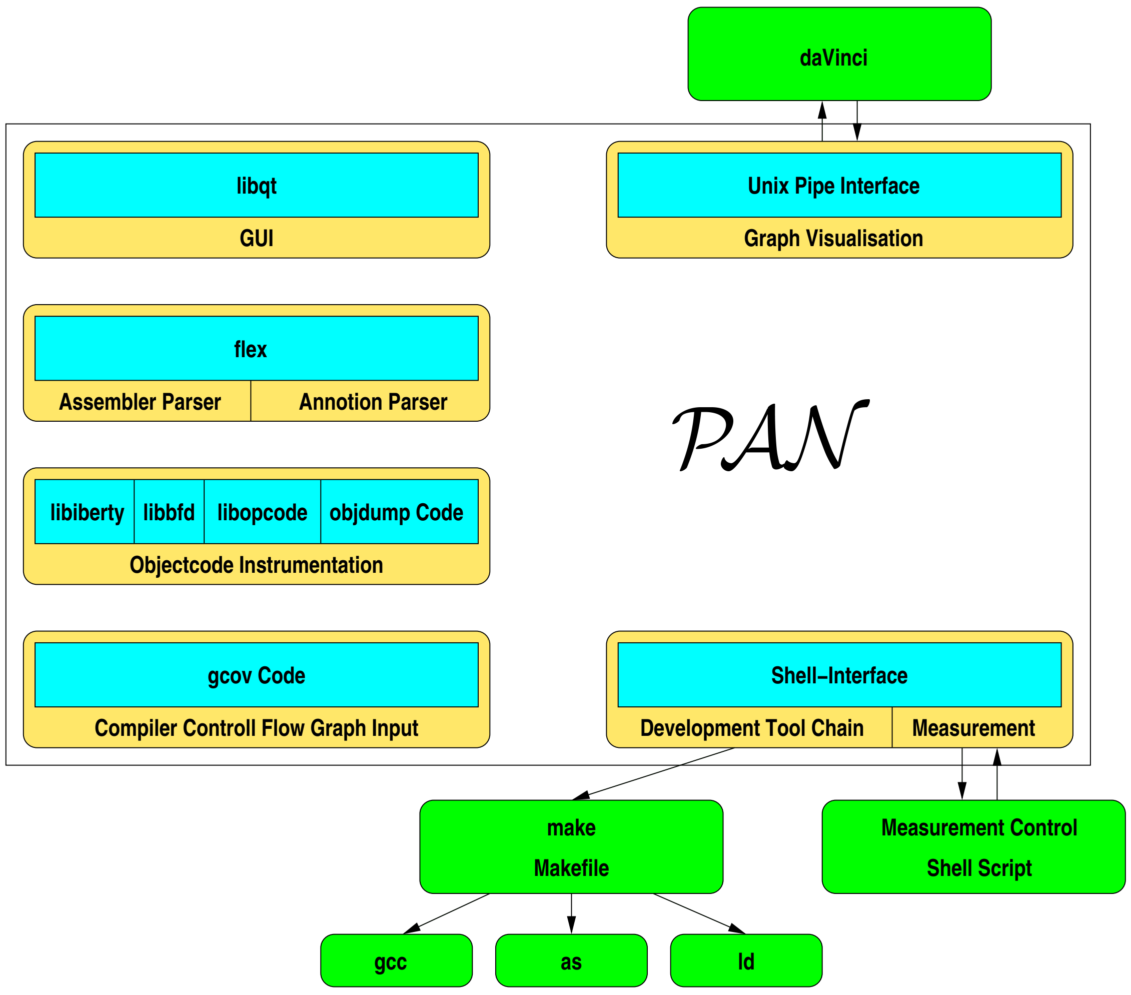


Figure 5.2: Overview of the structure of *PAN*

7. Annotations given on line of loop controlling structure.
8. Annotations for loops on line with loop-head.
9. No loop unrolling optimisation in the compiler.

Having all source code in one file simplifies the implementation of a tool considerably. A recursive implementation is only one aspect of a loop, and may easily be rewritten as an iterative approach. Typical implementation fields of such recursions are search algorithms within a linked list. Indirect function calls can not be resolved without simulating the assembler code and resolving the contents of the used registers and variables. The annotations are needed to bound the number of iterations of loops. To be able to identify the loop an annotation is for, it is requested that the annotation is given with the loop controlling structure. This leads to the restriction that no two loop controlling structures may be on the same line. Due to a quirk in the compiler, it is additionally necessary to provide an identical annotation at the head of the loop if the loop controlling structure in the source code is located at the end of the loop. The utilised compiler, `gcc`, uses very complex optimisation techniques. Most of these techniques can be ignored by utilising the assembler text for analysis. Loop unrolling is an optimisation approach that avoids branch instructions by concatenating several instances of the loop body, and reducing the number of iterations accordingly. To facilitate this, the loop is divided into a part detecting the number of iterations necessary, another part with a multiple of the loop body[1], and the final part executing the loop iterations which do not fit into the loop body. Unfortunately, the compiler information on loop unrolling is located in a separate file and, in some cases, the information on loop unrolling is rather cryptic. An additional parser would be necessary to detect the degree to which the loops are unrolled. As this feature is not essential to the described method, the integration of this has been postponed. However, manual loop unrolling is encouraged, as described in Section 5.6.

5.2.2 Development Tool chain

The development tool chain utilised for the approach has only minor modification and requirements, as compared to the “conventional” one. The tool chain consists of `GNU gcc`, `GNU binutils` and `GNU make`. The `gcc` has been slightly modified to emit the control flow information with instrumenting the code itself. To do this the option `-ftest-coverage` has to be used. The output of the debugging information is enabled by the `-g` option. Additionally, it should be noted that the loop unrolling feature has been disabled in the compiler for the tests.

The `make` and the `binutils`, i. e. , `as` and `ld` have been used without further modifications. The `Makefile` has to contain the following rules and options:

- gcc options: `-test-coverage -g`
- assembler file generation rule dependent on the source file
- executable generation rule dependent on the assembler

Besides these, a number of options are required by the RT-inux and Linux operating system, but since these do not derivate from the “conventional” setup and may change with future versions of the operating system, they are not explicitly mentioned here.

[1]The number of loop bodies unrolled is usually to the power of two.

5.2.3 $\mathcal{PAN}$ as Controlling Unit

The tool $\mathcal{PAN}$ has three main tasks. One is the analysis of the code under investigation, another the control of the development tools, and the third the manipulation of the objectcode for path enforcement and controlling the measurements. The analysis of the measurement results is currently supported by a range of small programms which have not yet been integrated into $\mathcal{PAN}$.

Another separation line can be drawn between architecture dependent and architecture independent part of the tool. To enable portability a well defined separation line between these parts is essential. In our case the architecture dependent part is limited to a simplified assembler parser, the object code manipulation routine and a configuration file which defines architecture properties for post processing of the measurement data.

To perform the object code instrumentation and deinstrumentation, a considerable amount of routines from the `GNU` `binutils` package have been used. In particular, the disassembler part of `objdump` on top of `libopcode`, `libbfd`, and `libiberty` as back end have been used. The `libopcode` describes the instruction set, and the addressing modes for a specific architecture. By utilising this, the architecture dependent part can be reduced to detecting the binary code of the opcodes searched for. This is usually limited to a few. All other opcodes relevant for the disassembly of the code, especially instruction boundary detection, can be abstracted by utilising `libopcode`. The binary format is handled by `libbfd`. This makes the tool independent of the chosen binary format. A few examples of such binary formats are coff, dwarf, elf or aout. Various support functionalities are combined in `libiberty`. To change as little as possible on the existing routines, this library has been added to the tool.

The parsers used to extract the control flow graph out of the assembler code and the annotations out of the source code, have been implemented by means of the `GNU` lexical parser `flex`. The control flow graph emitted by the compiler is imported utilising routines from the `gcov` tool, which comes as a profiling tool with the `gcc` compiler package. The graphical user interface is based on `QT` library, with the exception of the graphs which are displayed using `daVinci`, a specialised graph visualisation tool[2]. The control of the development environment (described in Section 5.2.2) and the measurements themselves is handled via a shell interface. This allows for maximum flexibility in the system under test.

[2]`http://www.b-novative.com/products/daVinci/daVinci.html`

5.2.4 $\mathcal{PAN}$ Analysis Part

The analysis can be separated into graph analysis, and measurement optimisation. Following the extraction of the control flow graph out of the assembler code, the graph theoretical technique *dominator tree* generation, well known within compiler development, is used. The dominator tree is generated out of the control flow graph. This is done by reordering the nodes of the control flow graph as follows: Node 6 of a control flow graph dominates node 8 if every path from the initial node of the control flow graph to 8 goes through 6. In the dominator tree each node (e. g. 8) is directly predecessed by that dominator which is the last dominator on any path from the initial node. Figure 5.3 shows a few CFG examples and their corresponding dominator trees. Further details regarding this technique can be found in the so called dragon book of compiler design [1].

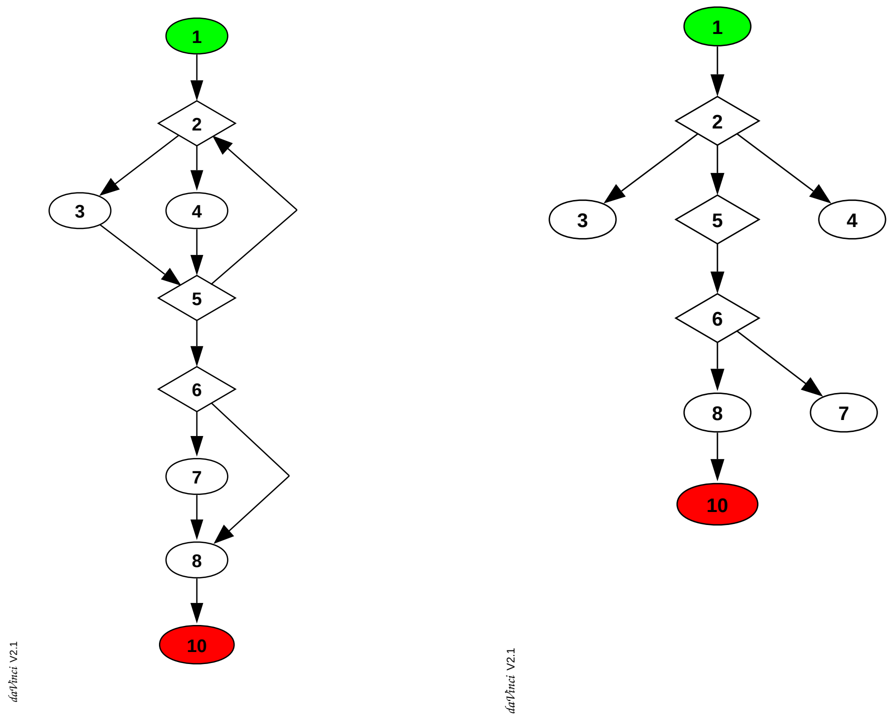


Figure 5.3: CFG Example and Corresponding Dominator Tree

Every edge in the control flow graph is checked, as to whether the target node of the edge would lead to a predecessor of the source node in the dominator tree. If this is the case, a backward edge of a loop has been identified. To simplify loop search for later analysis, each node is tagged with a nesting level. A loop body can be therefore searched by following the paths with the same, or higher nesting level than the loop controlling structure. In this step a reachability analysis is made as well. This includes, not only the code inside a single function but, starting from the main function of the thread (`thread_body()`), descends into all functions called with the exception of library function and RT-Linux system calls. These calls must either have no data dependent paths, or must contain the necessary measurement points during the measurements. Currently all registered functions must have the measurement points integrated, and are treated by $\mathcal{PAN}$ for analysis the same way as user profided measurement points.

In the next step, the annotations in the source code are evaluated. Until now the following annotations are identified:

`PAN_START` directs $\mathcal{PAN}$ to start the control flow graph analysis at this point in the code. This annotation has been added to avoid instrumentation of the boot up/preamble code of the RT-Linux threads.

`PAN_FIXED_LOOP` indicates that the loop annotated has a fixed number of iterations.

`PAN_VAR_LOOPCOUNT` denotes a loop with a variable number of iterations. The loop bounds are specified seperately by the following annotations.

`MIN <iterations>` provides the minimum number of iterations of the loop body.

`MAX <iterations>` provides the maximum number of iterations of the loop body.

`PAN_VARIABLE_PATH` specifies the loop to have input data dependent paths[3].

`PAN_FIXED_SPLIT` indicates that this branch instruction can not be touched for path inforcement as it is already otherwise covered.

The annotations `PAN_FIXED_SPLIT` and `PAN_FIXED_LOOP` are mainly used to avoid unnecessary instrumentation. Input data independent loops, with more than one path in the loop body are frequently found in, for example, digital filter algorithms. Where `PAN_FIXED_SPLIT` and a missing `PAN_FIXED_PATH` at the loop head avoid path enforcement, `PAN_FIXED_LOOP` avoids loop iteration enforcement.

The placement of measurement points corresponds to the requirements given in Section 4.3.6.3. First of all, these loops having the `PAN_VARIABLE_PATH` annotation set are instrumented with a measurement point. This measurement point is placed right after the label of the backward edge of the loop. Next, loops with fixed path, but variable iteration number are instrumented. With these loops the necessary place for the instrumentation of the loop control structure is inserted, and a measurement point is added directly after the loop exit. Now it is checked whether subroutines, which have input data dependent paths

[3]An inverted solution for this annotation (e. g. `PAN_FIXED_PATH` is possible, but brings no new aspects to the problem.

are called more than once inside a measurement block. If this is the case, a measurement point is set directly before all but the first call to this particular subroutine. Additionally in this step of assembler instrumentation, each branch instruction to be instrumented is preceded by two nop instructions, to allow for the path enforcement instrumentation of the objectcode.

According to the debugging information in the objectcode the address of the branch instructions, and calls to the measurement routines relative to the start of the file, is extracted out of the objectcode using code of `objdump`. This allows for fast measurement cycles.

Finally, the measurements need to be organised. To facilitate this, instrumentation graphs are built. The instrument graphs describe the topology of the paths to be instrumented. Each measurement point is the root of one of the instrumentation graphs. As previously mentioned, these measurement points may be either correspondingly tagged system calls and library functions, or user provided and automatically inserted calls to the measurement routine. The graph is acyclic and contains only branch nodes, measurement point nodes and additional subroutine nodes. All terminal nodes are measurement points. The instrument graph corresponds to the structure of the control flow graph between measurement points, with the restriction that all backward edges, all non branch and non measurement point nodes and all branch nodes that need no path enforcement, have been removed. The algorithm for measurement simply enforces all branches to be *taken*. The terminal node is used to traverse into the next instrumentation graph where this procedure is also done, until a full iteration of the RT-Linux thread body is completed. In the next measurement the last branch of the first instrumentation graph is set as *not taken*. Again the instrumentation graphs reached by this decision are triggered to do the path enforcement. The instrumentation graphs retain the information of the last selected path in a stack structure, and will always pick the next path to be measured when reached. In this way, the paths within different instrumentation graphs can be measured with maximum efficiency.

Calls to subroutines which are called at several places, and loops with data dependant paths in the loop body must be considered seperately. Both have the property of appearing in multiple instances in the instrumentation graphs. In the case of a loop body which needs instrumentation, it is part of at least three instrumenation graphs.

- One or more instrumentation graphs entering the loop body which ends at the measurement point in the loop body.
- A single instrumentation graph representing the loop body.
- One graph leaving the loop body, starting with the measurement point in the loop.

With subroutines calls, the scenarios depicted in Figure 5.4 have to be dealt with. The

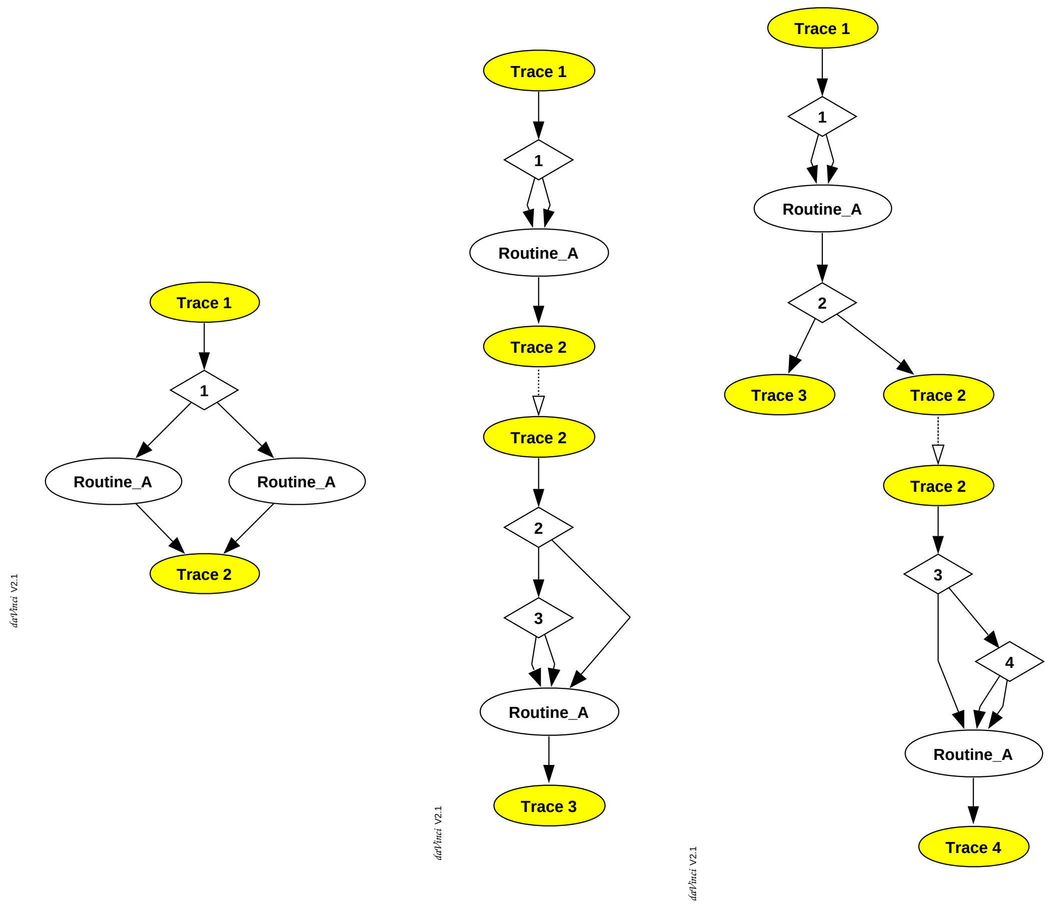


Figure 5.4: Possible Scenarios for Multiple Calls of a Subroutine

dotted transition between two trace nodes indicates a transition from one instrumentation graph to another. To achieve the following results critical and uncritical cases need to be identified:

- Measurement of all path combinations
- Avoidance of excessive measurements
- Avoidance of overpessemistic results

Calls in disjunct paths can be considered as uncritical. This case includes subroutines which are called only once. The structure of the called function is integrated into the instrumentation graph of the calling function.

More critical are those calls in conjunct paths, which need a more detailed handling. As previously described, the placement of measurement points ensures that calls of this kind are located in different measurement blocks. An inclusion into the instrumentation graph of the calling function would lead to multiple instrumentation instances of these calls. Thus a change in one instrumentation graph would directly interfere with the paths in all other instrumentation graphs. One solution, avoided in this case due to efficiency reasons, would be to measure each instrumentation graph on its own. A more feasible solution would be the use of separate graphs (referred to as subroutine graphs from here on) to monitor "reused" functionalities. The subroutine graphs change their paths independently of the normal instrumentation graphs. The progress of these path instrumentations is documented by a counter, which is reset to zero whenever all paths have been enforced. With the subroutine graphs, special nodes are inserted at the point of the calling instrumentation graphs. This special node ensures that the instrumentation graph in which it is contained does not change for a full cycle of the subroutine graph. Thus, it is ensured that all path combinations are measured. However, if an instrumentation graph contains more then one subroutine graph in a conjunct path, only one subroutine graph changes at the normal rate, while the others are scaled down to ensure once again that all path combinations are measured.

Furthermore, with regards to "reused" functionality, loop bodies containing one or more measurement points face a similar problem. Loop bodies are split up in two parts. The first part is associated with all paths entering the loop body and the instrumentation graph of the loop body itself, while the second part belongs to the instrumentation graph of the loop body and the instrumentation graph of the leaving path. Due to the fact that both parts share functionality with at least two instrumentation graphs, the same problem, but also the same solution as for the calls in conjunct paths applies.

In order to avoid the generation of subroutine graphs for many loops, it has been decided that for loops needing path instrumentation, the measurement point will be placed just after the loop head. In this way, most loop bodies have the relevant alternatives only in the instrumentation graph of the loop body itself, and the leaving edge. In this way, all potential paths of the loop body instrumentation graph are automatically measured. The path is enforced according to the necessities of the instrumentation graph of the leaving path, and the loop body instrumentation graph is enforced as a byproduct.

5.3 Test Setup

In the tests shown the focus is set on the AMD Athlon architecture. The Intel Pentium III processor is shown in a few examples to illustrate the applicability of the approach.

5.3.1 Athlon Hardware

To facilitate the tests of the application software on the AMD Athlon architecture, two systems equipped with an Athlon processors with model id 4 and stepping 2 have been used. In one system the processor was clocked with 800 MHz, in the other with 900 MHz. Otherwise the systems were identical. The 900 MHz system was deployed for the measurements on the RT-Linux operating system while the 800 MHz Athlon was used for the measurements on the application software. The descriptive measurements as regards memory load and branch prediction have been also taken on the 800 MHz system. As it is, this data has to be collected for each system individually and the given values are only to provide an overview.

Both processors provide 256 KByte of second level cache, and 64 KByte for each of the level 1 data and instruction cache. The branch prediction unit has been measured so as to need 13 additional cycles for a misprediction. This data was measured by providing the worst case of an indirect jump instruction directly preceded by the computation for a pointer to the jump-table, thus ensuring the maximum latency possible. The test has been repeated in 1000 cycles of 2^{16} iterations of the test loop, and compared to a repeated sequence of branches to the identical target, thus enabling the usage of the branch target buffers.

The load of a cacheline from main memory into the L1 data cache has been measured so as to be 209 CPU cycles for the 800 MHz hardware. The measurements were taken with no external bus activity, and under worst case conditions i. e.; each cacheline is only touched once, and only two opcodes on register values are executed. It has to be noted that these measurements have to be repeated if another hardware is used, since this data not only depends on the processor, but also on bus and memory latencies.

Finally, the amount of time consumed by the measurement routine itself, outside the two timestamps, is of relevance. To determine this time, two consecutive measurement points have been utilised. The measurements indicated the additional time consumption as being between 400 and 672 cycles.

5.3.2 Pentium III Hardware

A Pentium III Katmai system with 500 MHz, served as testbench for the Intel P6 architecture. The L2 cache is, with 512 KByte, twice the size of the Athlon processor but runs only on halve speed. To get the parameters of the processor, similar tests as were used the AMD Athlon were performed. A missprediction in the branch prediction unit also needs 13 cycles. A cacheline load on the Pentium III Hardware was only 72 cycles. This matches with the description of 14 front side bus cycles for an L2 cache miss (cf. Section A.1.3 on page 128). The difference to the Athlon architectiure may be explained on one hand by the larger difference between front side bus and CPU frequency and the double amount of data fetched due to the larger cacheline size of the Athlon processor. This results especially in the write back of the preempted cache contents problems.

5.3.3 Operating System

The operating system chosen for the measurements is Linux version 2.4.1, with RT-Linux version 3.0 extension. The main change necessary to the operating system was introduced in the memory allocation algorithm for the modules. In order to achieve reproducible results, a fixed memory portion was allocated at boot time to hold RT-Linux and it's application thread modules. The modules were always loaded in identical order into the kernel, to ensure identical memory usage. The RT-Linux main module has been extended to provide the routines necessary for the measurement. Additionally, a module providing the interface to read out the test results is necessary. The discussion of actual measurements is limited to the interrupt handling and the scheduler as prominent part in RT-Linux, both taken on an AMD-Athlon processor clocked with 900 MHz (cf. [81]).

5.3.3.1 Interrupt Management

As regards the interrupt management two main scenarios have to be considered. In one case the time necessary until the corresponding interrupt service routine is called, on the other case the time needed to note down a interrupt which will be later resolved by the Linux operating system. The routine `rtl_intercept()` is the main source of execution time due to the access to the PIC to acknowledge the interrupt. Only with this acknowledge a further interrupt will be accepted by the PIC. The focus of the measurements is therefore set on the `rtl_intercept()` routine.

Figure 5.5 shows the WCET of `rtl_intercept()` and the corresponding kernel density, until the interrupt service routine for this particular interrupt is entered. The minimum and maximum time needed for this operation is 5669 cycles and 7348 cycles. Additionaly Figure 5.5 shows a quite typical characteristic of hardware accesses on PCs. Several equidistant peaks indicate the external hardware latency.

Similare results can be seen in Figure 5.6 which depicts the time needed to delay an interrupt, which is served when no thread of RT-Linux is ready. The observed execution times lay between 7354 and 9618 cycles. In this case a few additional main memory accesses, to tag a specific interrupt as occured, lead to the fact that the execution times between the peaks induced by the hardware accesses merge into each other.

The change of privilige level necessary if the system is in user mode when the interrupt request occurs takes between 18000 and 22000 cycles.

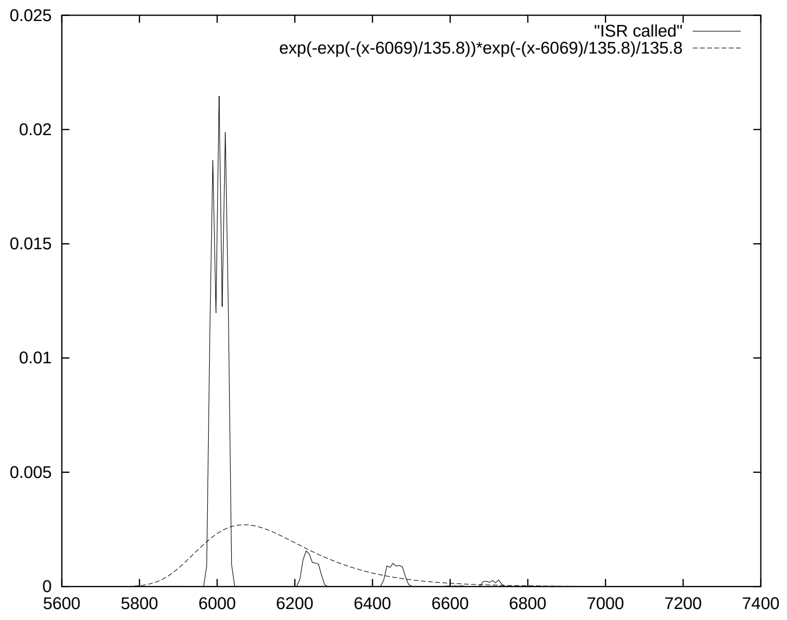


Figure 5.5: Time Needed by `rtl_intercept()` to Call the ISR

5.3.3.2 Scheduler

The rescheduling process can be divided in three parts. The first is the occurance of an interrupt, until the interrupt service routine is called plus the execution time of the interrupt service routine. The first part has been investigated in the previous Section. The time for the timer interrupt service routine, used in the experiments was between 8866 cycles and 11093 cycles, with a mean of 9703 cycles and a deviation of 157 cycles.

The second time to be reasoned about is the scheduling algorithm itself. This has a number of parameters. First of all the number of threads to be scheduled. Each thread added to the system results in a rise of approximately 350 cycles in the time needed by the scheduler. The RT-Linux scheduler is built that way, to check all RT-threads if they are ready. Thus the execution time does not depend on the position of a thread in the thread list. However, the thread influences the scheduling time, by indicating whether the floating point context needs to be saved or not. Each thread using floating point operations must explicitely state so in the initialisation phase. If such a thread enters the running state, the floating point unit context is saved. This operation takes between 1474 and 2222 clock cycles.

If the tasks are scheduled using the *one shot mode* of the RT-Linux time management, the timer has to reprogrammed. Figure 5.7 shows the kernel density, and the corresponding extreme value distribution of the small routine, programming the timer described in

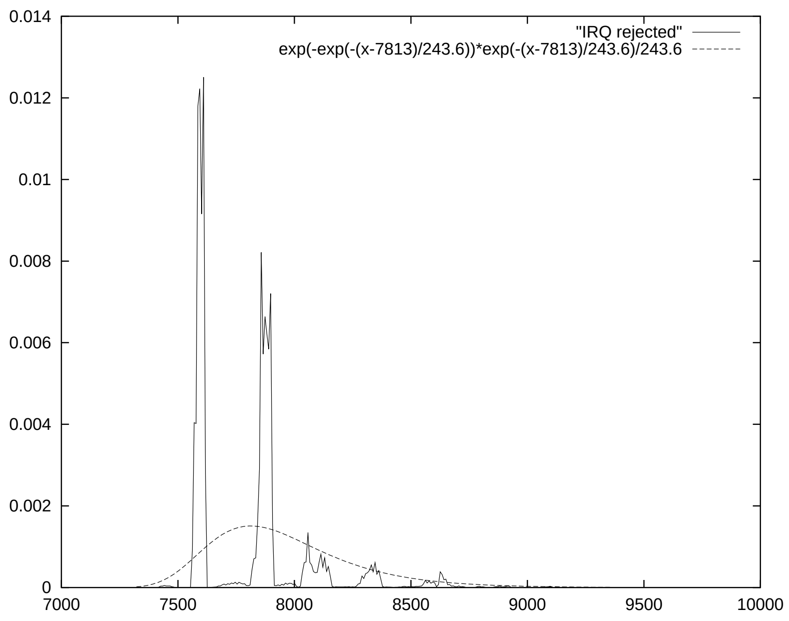


Figure 5.6: Time Needed by `rtl_intercept()` to Delay an Interrupt

Section 4.2.2.2. The minimum time needed for this routine was 3499 cycles, while the largest value was 4532 cycles.

The typical hardware latencies of the control bus can be identified in Figure 5.7. Table 5.1 provides an overview on the approximate values for the different parts of the scheduler.

Parameter	Execution Time
IRQ to start of scheduling algorithm	10.4 *μs* if already in kernel mode, + 25 *μs* if in user mode
number of threads in the list	0.4 *μs* for each thread
FPU context switch	1.8 *μs*
reprogramming of timer	3.9 *μs*
cleanup delay till thread actually starts execution	2 *μs*

Table 5.1: Approximate Values for Scheduling on 900 MHz Athlon [81]

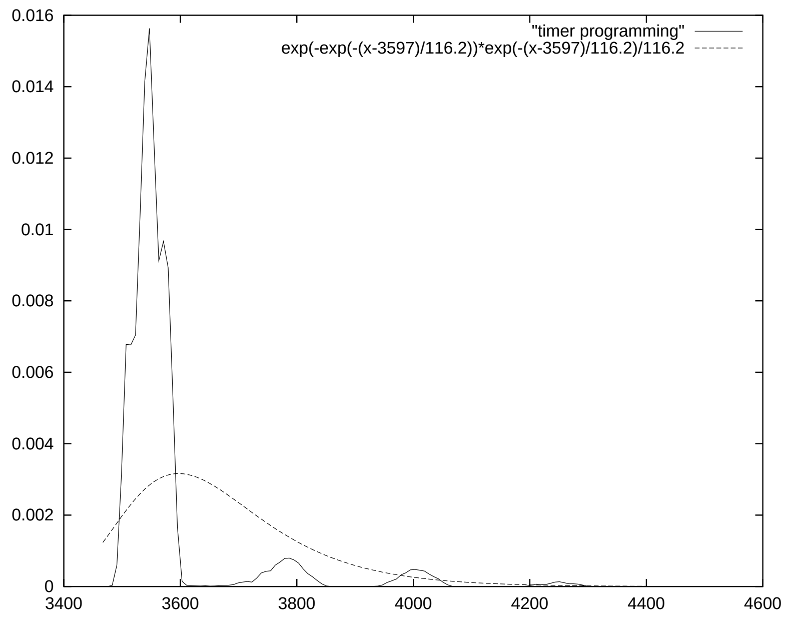


Figure 5.7: Programming the Intel 8254 Timer

5.4 Example Applications and Evaluation

The applications used to test the approach were inspired by the algorithms utilised by the *Embedded Microprocessor Benchmark Consortium (EEMBC)*. Not all EEMBC algorithms are suitable for processors like the Intel Pentium III or the AMD Athlon like, for example, the ignition timing used in a car engine, since the execution jitter exceeds the execution time of the code.

A simple number crunching algorithm, like a fast fourier transformation, incurs no problem for the presented approach. The path through the code and the number of loops is not input data dependent, assuming that the length of the vector is invariant. In most applications the vector length is fixed, or at least only a small number of vector lengths are used as input. In this case all possible lengths of the input vector have to measured separately, and applied as appropriate. In the measurements a vector length of 4096 entries was chosen.

Similar results are achieved for matrix multiplication, discrete cosinus transformation or dithering of images. The matrix multiplication was operated on matrices with 128 columns and 128 rows. A image of 96*96 pixles was taken as source for the discrete cosinus transformation with blocksize 8. The dithering was also done on a 96*96 pixel image. In this case the placement of additional measurement points could be avoided by rewriting a critical section and replacing an `if/else` statement by bit manipulations.

Algorithm	Mean	Deviation	Min	Max
FFT	$1.97 * 10^6$	256	1964842	1970123
matrix multiplication	$1.55 * 10^7$	184	15555140	15556459
discrete cosine transformation	$7.51 * 10^6$	140	7514004	7514530
image dithering	$5.18 * 10^6$	61	5175933	5176493
viterby soft decision	$2.82 * 10^4$	155	28205	31982
viterby soft decision (recoded)	$2.35 * 10^4$	39	23595	23814

Table 5.2: Results of Sample Applications on Pentium III System in Processor Cycles

Algorithm	Mean	Deviation	Min	Max
FFT	$9.50 * 10^5$	3815	912890	972475
matrix multiplication	$1.25 * 10^7$	4702	12451533	12527198
discrete cosine transformation	$4.52 * 10^6$	4611	4452611	4522123
image dithering	$3.51 * 10^6$	4877	3436897	3516690
viterby soft decision	$2.17 * 10^4$	1020	18312	28389
viterby soft decision (recoded)	$1.63 * 10^4$	427	14035	16843
bezier algorithm	$2.42 * 10^7$	5328	24200340	24229748

Table 5.3: Results of Sample Applications on AMD Athlon 800 MHz System in Processor Cycles

An application used in the mobile communications world is the viterby soft decision algorithm [84]. Here, the aspect of intelligent coding can be shown. While simple implementation loops over the bits of an input string and checks whether a given bit is set and provides corresponding results, the second implementation checks the bits by means of an & operator and sets the result without needing a branch instruction. This modification of the algorithm speeds up the algorithm not only in the provided results, but also in real life. As Table 5.2 and Table 5.3 show, the Athlon architecture is considerable faster than the Intel Architecture. However, the Intel architecture has the advantage of smaller deviations in the execution time.

Typical examples in the image processing field are image filtering, or the computation of Bezier curves which have at least some paths to be enforced. The Bezier curves, as described in [6], are a good example of how to obtain a reliable bound on the execution time, while avoiding too much overestimation by using unconventional coding solutions, as described in Section 5.6. The algorithm computes the parameters in such a way, that start and endpoint are met. After that, the algorithm loops over a given number of steps, between the start and the endpoint, and computes the location of the points to be coloured black. Unfortunately, this point has to be checked against being out of bounds. By simply taking the algorithm as it is, the malus introduced by the approach would be enormous. The algorithm has been rewritten in order to avoid this excessive overestimation.

Due to the many manual operations on the measurement results which are not yet integrated into $\mathcal{PAN}$, the following two examples have only been tested on the Athlon architecture. The image filtering algorithm illustrates a major drawback of the approach. The algorithm heavily depends on cache usage and is in itself very small. The initial tests with the algorithm as is resulted in an overestimation of about factor 20. Then the loops were differently partitioned to avoid the excessive negative effects in the caches and the measurement was repeated. The code needing instrumentation was moved in a seperate very compact loop which was additionally unrolled 3 times[4]. The changes in the code were only straightforward and in no way exotic. In average the code without instrumentation took $4.82 * 10^6$ cycles with a standard deviation of 6722 cycles. The instrumented code consisted needed 2 additional measurement points in loops, which resulted in 8000 measurement blocks. The join operation according to Section 4.6.3 resulted in a mean value of $10.2 * 10^6$ cycles and a standard deviation of $17.0 * 10^3$ cycles. This shows that the overestimation can be kept in close limits by following simple considerations. An avoidance of the additional measurement points as has been done with the viterbi and bezier algorithm is still possible and would result again in a single path through the program.

The largest application under test is the codebook excited linear predictor, or short celp algorithm. The code is derived from version 3.2c of the DoD's Federal-Standard-1016 implementation of the lossy speech compression algorithm. After most of the comments of the original have been removed, the algorithm has approximately 3000 lines of code and 140 annotations. The code has been rewritten in some places, in accordance with the recommendations given in Section 5.6. 31 measurement points were automatically introduced by the placement algorithm according to the rules provided in Section 4.3.6.3. In order to manage the complexity 4 additional measurement points were introduced manually. The analysis of the code led to 56581 measurement blocks. The analysis of the execution count of the measurement blocks has been done by hand using additional information about non rectangular loops which cannot be expressed by the simply annotation scheme. The deinstrumented version took $2.23 * 10^8$ cycles. Compared to this the measured mean was $4.30 * 10^8$ with a standard deviation of $5.62 * 10^6$. This result was achieved by analysing measurement results, identifying bottle necks and modifying the code to minimise the number of measurement points necessary. To compare the amount of overhead introduced by the `nops` left after deinstrumentating the code, a version without these nops has been tested and shown $2.19 * 10^8$ cycles.

[4]The image investigated was coloured with three colour components.

5.5 Limitations to the Instrumentation

Obviously, the enforcement of paths for the measurements has some critical issues. One is that some loops, especially in search algorithms, terminate when no more members of an array or a linked list need to be processed. Typical examples of such an algorithm are search algorithms, which search for the largest member in a linked list of data objects, where the end of the list is marked by a `NULL` pointer, or a string processing algorithm with variable `NULL` terminated string length. Enforcing the maximum number of loop iterations, without regard for the input data, would simpy lead to a segmentation violation during the measurement. With these a set of input variables must be chosen, in such a way that these requirements are fullfilled.

A second problem is code that is guarded by `if` conditions to avoid divisions by zero, or segmentation violations. A typical example of this is the previously desribed Bezier algorithm as in it's "unmodified" form. In this case, those pixels, located outside the memory area used for visualisation, are simply discarded. With the Bezier algorithm it is comparably simple to provide data which does not leave the memory area allocated for the display, but for more complex algorithms this is not so easy to achieve, especially when the problematic code lies in nested loops. Depending on the code, alternative solutions to this problem may be chosen. Either the input data is changed in such a way that all paths are valid, or it is checked which path combinations are naturally checked by input data, and the corresponding branch instruction is annotated with `PAN_FIXED_SPLIT`.

5.6 Coding Guidelines

Two main reasons drive the coding style for this approach. On the one hand the approach has to minimise the number of measurement blocks, while on the other hand it is favourable to minimise the deviation of the execution time and improve the performance of the code.

To avoid of additional measurement points in the execution path, and therefore measurement blocks, the rules for applying additional measurement blocks in Section 4.3.6.3 must be investigated:

1. Subroutines with more then one path called from more than one point within a measurement block.
2. Loops with more than one path in the loop body.
3. Lookup-Tables.

The problem of subroutines, which are called from more the one point, can be solved by duplicating the functionality, and using separate instances for the separate calls. Implementations supporting this are inline functionality in the declaration of the code, macro definitions or copying the code and changing the identifier. The first solution is the cleanest one, but one has to check whether the compiler really implements this feature.

To facilitate loops with alternative paths, it must be checked if another, maybe suboptimal implementation can avoid the necessity of alternative paths. A rather unconventional

```
for (count = 0; count < 100; count++) {
  if(x[count] < 0) {
    x[count] = -x[count];
  }
}
```

```
int helper[] = { -1, 1 };
for (count = 0; count < 100; count++) {
  x[count] = helper[(x[count]>>31)+1]*x[count];
}
```

Figure 5.8: Sample Code for an Alternative Loop Body

example to avoid alternative paths inside a loop body is given in Figure 5.8, where the norm of a number of array values is computed. A short analysis on the assembler code level, compiled from the sourcecode with comparably simple optimisation[5] reveals that the loop body of the first option takes either six or eight assembler instructions, depending on whether the inversion is actually computed or not, while the second option utilises seven instructions. In the face of mispredictions of the branch prediction, and the necessary rollback, it can even be expected that the second, more complicated version, is considerably faster than the simple straighforward implementation. The optimisation guides of AMD Athlon [4] and Intel Pentium III [40] encourage the avoidance of branch instructions to accelerate execution. However, such “virtuous” changes in the code make it less human readable and less portable. The first can be mended by excessive use of comments, but the loss in portability has to be carefully balanced against the gain, of closer bounds on the execution time.

Another possibility for reducing the number of measurement points encountered for a specific functionality, is the manual unrolling of a loop. Since the performance advantage of caches for loops is voided by the introduction of an additional measurement block, as regards performance the unrolling incurs no disadvantage compared to the compact loop. Nevertheless, it has to be noted that the number of paths to be searched rises considerably. This in turn may lead to a greater number of measurements necessary. The loop unrolling is even possible in a search algorithm, where it is impossible to compute

[5] `gcc` optimisation `-O2`

a priori the number of iterations of the loop even though an upper bound exists. Sample code is depicted in Figure 5.9.

```
count = 0;
do {
  retval = do_something(x[count]);
  count++;
} while(retval != search);
```

```
count = 0;
do {
  retval = do_something(x[count]);
  count++;
  if (retval != search){
    retval = do_something(x[count]);
    count++;
  }
} while(retval != search);
```

Figure 5.9: Sample Code for Manual Loop Unrolling

In general, lookup-tables are prohibited due to the indeterministic memory access. A workaround can be achieved by preloading the complete lookup-tables into L1 data cache. With AMD Athlon and Intel Pentium III processors, the MMX `PREFETCH` opcode can be used to achieve this. To validitate this, two restrictions must be considered:

1. The prefetch of the data has to be initiated in advance, in such a way, that the data is definitely present when accessed.

2. The data has to fit into L1 data cache memory, together with all data needed between the prefetch and the lookup.

The first restriction requires an excellent knowledge of the hardware, and a series of tests to determine the amount of code to be placed between prefetch and lookup while the second can be checked easily.

Usually this technique is only suitable for lookup-tables which are small compared to the L1 cache, otherwise the gain of the lookup-tables would be voided due to the load time of the `PREFETCH` opcode. On the other hand, the processor bus shows its peak performance during burst access to main memory. The tradeoff must be balanced as the case arises.

5.7 Real-Time Analysis

The approach of Lee et al., presented in [51] focuses on computation of the additional preemption delay, by intersecting cachelines of preempted and preempting threads. In order to perforn this, a detailed analysis of possible schedules is made, which is then solved by an integer linear programming approach. Since this approach provides an integrated approach for WCET and schedulability analysis, and is limited to instruction cache effects, the adaption to uptodate processors is nearly impossible.

A quantitative comparison to some degree is only possible with the work of Busquets et al. in [16]. The limitation is caused by the fact that in handling branch prediction it is nearly impossible to bound the effect on the branch prediction solely by analysing the preempting thread. Thus, the comparison is limited to the effect of caches and scheduling.

As in [16] the PN series of the hartstone benchmark [83] are taken as the basis for developing the comparison test. The hartstone benchmark has been developed by Weiderman and Kamenoff and provides a real-time systems performance benchmark. The PN series is a set of 5 periodic threads with different, non-harmonic inter-arrival times. Due to the age of the test, the processor performance given in the benchmark does not correspond with modern processors, and some relevant data is not specified. To perform the test no code is executed - both schedulability analysis are simply exercised.

Since the blocking time is equivalent in both approaches, it is not separately considered. Additionally, the approach presented in this work will be used with a single triggering tuple, in accordance with Section 4.1, thus reducing it to simple periodic threads, and making the approaches compatible for comparison. The time needed to load the number of cachelines the thread utilises is specified with the term *cache load*. The value *scheduling penalty* contains the time needed for a thread switch, and a penalty for the interference of the other acceleration techniques (e. g. branch prediction or pipeline).

Table 5.4 provides the basis for the following investigation. Equation 2.2 of Busquets et al. in the extrinsic interference γ_j is comprised of the sum of the cache load. The scheduling penalty for the approach in this paper, the penalty δ_i, is given by:

$$\delta_i = schedulingpenalty_i + cacheload_i * usabilityfactor \tag{5.1}$$

It has to be noted, that this method is only used for comparison of the approaches. The $usabilityfactor$ models, in this case, the amount of useful contents in the cache. Due to the fact that the application is usually split in several measurement blocks the amount of useful contents in the cache will be small when compared to the amount of data and code loaded into cache during the execution of the application. In the final application, δ_i has to be provided by the WCET analysis by analysing, not only the cache usage, but also the other acceleration techniques as described in Sections 4.3.7 and 4.4.7. Figures 5.10 a)

Thread	Frequency [Hz]	Period [ms]	Computation Time [ms]	Cache Load [ms]	Scheduling Penalty[ms]	Test System
1	31	32.26	2	0.50	0.10	1
2	17	58.82	4	1.00	0.11	1
3	12	83.33	5	2.50	0.11	2
4	10	100.00	7	2.75	0.12	2
5	7	142.86	9	3.25	0.13	1,2
6	6	166.66	10	4.25	0.14	2
7	5	200.00	13	5.25	0.14	1,2
8	3	333.33	21	2.25	0.16	1

Table 5.4: Sample Thread System

and b) give an overview of the response times of the sample systems threads with varying usability of the cache contents. The dashed lines corresponds to the response time of the threads, computed with the method of Busquets et al. [16]. In Figure 5.10 a) only the thread set consisting of the threads 1, 2, 5, 7 and 8 is considered, which corresponds to the setting in the hartstone PN series.

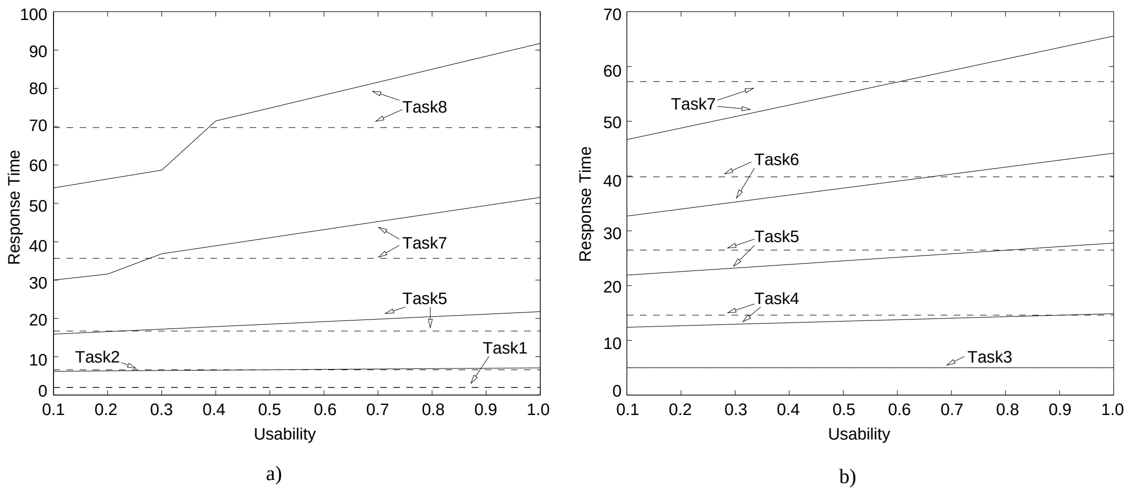


Figure 5.10: Response Times of the Sample Thread Systems with Varying Usability

It can easily be seen, up to a usability of 30 % of the contents of cache, that the new approach is superior for the given thread system, while the approach degrades when a utilisation of more then 40 % is reached. Analysis of a number of programs have shown that a typical amount of useful data in the cache is about 30 to 40 percent.

In the second test system only threads 3, 4, 5, 6 and 7 are taken, thus simulating a set of more uniform requirements. Figure 5.10 b) depicts the response times for this set of threads. With this system the break even point of the two approaches is reached later at approximately 70 %.

In the presented approach the usability of the cache is less, due to the method of estimating the WCET. The decision as to which of the two methods is superior also depends on the method of WCET estimation, which has to provide the data regarding cache usage.

6 Conclusion

The main objective of this theses was to show the applicability of a measurement based approach, to estimate the worst case execution time on up to date processors. The problem with modern processors is their built in indeterminism. The analysis of the Intel Pentium III and AMD Athlon reveals that an approach with a cycle true simulator of the processor is infeasible, due to the various acceleration techniques. At this point the question arises as to why two such badly built processors were picked as application example. The reasons are twofold. On one hand these processors belong to the mass market, and are surely to appear in different disguises in many embedded systems, which by nature are often real-time systems, if not allways in the classical sense of meaning. The second reason is that the techniques found in these processors, which cause numerous problems in worst case execution time estimation, are found more often in more typical embedded processors. In this way caches have found their way into various processor cores, and branch prediction is not far behind. An approach that provides good results on misbehaving processors, will most likely provide also good results on processors having not all, but a few of the fancy acceleration techniques.

To estimate the WCET, measurements on the hardware are taken. Prior to the start of the measurements, the execution and acceleration units are set as far as possible in the worst case state. The part of the execution and acceleration units, that can not be reliably enforced into a given state, or where the worst case state is unknown, is randomised, where appropriate, or covered by additional penalties added to the measured execution time. By enforcement of the paths taken in the program, a reliable bound on the WCET is possible. Additionally measurements can be deployed to bound the preemption delay in such processors. The enforcement of paths in subroutines and loops make it necessary to partition the program under test into several measurement blocks which are seperately measured. Thus it is possible to ensure that all path combinations are covered. In algorithms depending heavily on caches, the overestimation induced by measurement points in critical small loops is enormous. This overestimation can be vastly reduced, by using the coding rules described in this thesis. In addition to the measurement of application software the analysis and measurement of the execution time of operating system functionality has been shown.

As a means to provide a value for the confidence in the aquired results, the extreme value statistics by Burns and Edgar (cf. [14]) approach is utilised. The approach has been extended to be able to compute the extreme value probability density function of a whole task consisting of several measurement blocks. Finally a scheduling analysis algorithm is provided, that is capable of handling the additional time induced by the disruption of the working set in the acceleration techniques of the processor whenever the task is preempted.

The presented approach of estimating the execution time by utilising measurements can be improved in various aspects. The current version of annotations is very simplistic and has been only implemented to provide the absolute necessary features to make the measurement based approach possible. The annotations can be extended to a more advanced state and partly eliminated by using a localised symbolic execution unit. An overall symbolic execution to provide loop bounds and detect infeasible paths will likely be to complex.

A further extension to the approach is to use the results of the path enforcement in order to bound the deviation of the different paths. This is especially usefull in small loop bodies. In this case the deviation can be utilised to add a penalty on the execution time measured of the uninstrumented loop. Thus a closer bounding of the WCET is possible, while still retaining considerable control of execution paths. This can be supported by a more extensive use of statistics.

Bibliography

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Priciples, Techniques and Tools*. Addison–Wesley Publishing Company, 1986.

[2] ALLEN, A. O. *Probability, Statistics, and Queueing Theory*. Academic Press, Inc., 111 Fifth Avenue, New York, 1978.

[3] AMD. *AMD Athlon Processor Architecture*. Sunnyvale, CA, USA, Aug. 2000.

[4] AMD. *AMD Athlon Processor x86 Code Optimization Guide*. Sunnyvale, CA, USA, Sept. 2000.

[5] AMD. *Whitepaper AMD Athlon Processor and AMD Duron Processor with Full-Speed On-Die L2 Cache*. Sunnyvale, CA, USA, June 2000.

[6] ANAND, V. B. *Computer Graphics and Geometric Modeling for Engineers*. John Wiley & Sons, Inc., 605 Third Avenue, New York, 1993.

[7] ARNOLD, R., MÜLLER, F., WHALLEY, D., AND HARMON, M. Bounding worst–case instruction cache performance. In *Proc. of the IEEE Real–Time Systems Symposium (RTSS'94)* (Dec. 1994), IEEE Computer Society Press.

[8] ATANASSOV, P., KIRNER, R., AND PUSCHNER, P. Using real hardware to create an accurate timing model for execution–time analysis. In *Proceedings of the 1st Real–Time Embedded Systems Workshop (RTES'01)* (London, UK, Dec. 3 2001), IEEE.

[9] BARABANOV, M. *A Linux-based Real-Time Operating System*. M.S. Thesis, June 1997.

[10] BASUMALLICK, S., AND NILSEN, K. Cache issues in real–time systems. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real–Time Systems (LCTRS'94)* (Orlando, FL, USA, June 21 1994).

[11] BLIEBERGER, J., FAHRINGER, T., AND SCHOLZ, B. Symbolic cache analysis for real–time systems. *Journal of Realtime Systems 18* (2000), 181–215.

[12] BURNS, A. Preemptive priority based scheduling: An appropriate engineering approach. In *Advances in Realtime Systems*. Prentice–Hall International, Inc., 1994, pp. 225–248.

[13] BURNS, A., AND EDGAR, S. Predicting computation time for advanced processor architectures. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems* (Stockholm, Sweden, June 19–21 2000).

[14] BURNS, A., AND EDGAR, S. The use of extreme statistics to predict computation time for advanced processor architectures with branch prediction. Tech. rep., University of York, United Kingdom, Department of Computer Science, Real-Time Research Group, 2000.

[15] BURNS, A., AND EDGAR, S. Statistical analysis of WCET for scheduling. In *Proc. of the IEEE Real–Time Systems Symposium (RTSS'01)* (London, United Kingdom, Dec. 4–6 2001).

[16] BUSQUETS-MATAIX, J. V., GIL, D., GIL, P., AND WELLINGS, A. Techniques to increase the schedulable utilization of cache-based preemptive real-time systems. *Journal of System Architecture 46* (2000), 357–378.

[17] BUSQUETS-MATAIX, J. V., WELLINGS, A., SERRANO-MARTIN, J. J., , ORS-CAROT, R., AND GIL, P. Adding instruction cache effect to exact schedulability analysis of preemptive real-time systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems* (L'Aquila, Italy, June 10–12 1996), IEEECSP, pp. 271–276.

[18] BUSQUETS-MATAIX, J. V., WELLINGS, A., SERRANO-MARTIN, J. J., , ORS-CAROT, R., AND GIL, P. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of IEEE Real-Time Technology and Applications Symposium* (Boston, June 12–14 1996), IEEECSP, pp. 204–212.

[19] CHAPMAN, R., BURNS, A., AND WELLINGS, A. Integrated program proof and worst–case timing of SPARC Ada. In *Proceedings of the ACM SIGPLAN Language, Compiler, and Tool Support for Real-Time Systems (LCTS) workshop* (Orlando, Florida, June 1994), ACM Press.

[20] COLIN, A., AND PUAUT, I. Worst case execution time analysis for a processor with branch prediction. *Journal of Realtime Systems 18* (2000), 249–274.

[21] COLIN, A., AND PUAUT, I. Worst case execution time analysis of the RTEMS real–time operating system. In *Proceedings of the 13th Euromicro Workshop on Real-Time Systems* (Delft, Netherlands, June 13–15 2001).

[22] ERMEDAHL, A., AND GUSTAFSSON, J. Deriving annotations for tight calculation of execution time. In *Euro-Par'97, 20th Workshop on Real-Time Systems and Constraints* (Passau, Germany, Aug. 1997).

[23] EVERS, M., AND YEH, T.-Y. Understanding branches and designing branch predictors for high performance microprocessors. *Proceedings of the IEEE 89*, 11 (Nov. 2001), 1610–1620. Special Issue on Microprocessor Architecture and Compiler Design.

[24] FERRARI, D. *Computer Systems Performance Evaluation*. Prentice–Hall International, Inc., Englewood Cliffs, NJ, 1978.

[25] FERRARI, D., SERAZZI, G., AND ZEIGNER, A. *Measurement and Tuning of Computer Systems*. Prentice–Hall International, Inc., Englewood Cliffs, NJ, 1983.

[26] FISHER, R. A., AND TIPPET, L. H. C. Limiting forms of the frequency distribution of the largest and smallest member of a sample. *Proceedings of the Cambridge Philosophical Society 24* (1928), 180–190.

[27] FOG, A. Branch prediction in the pentium family. *Dr. Dobb´s Journal* (1998). http://x86.org/articles/branch/branchprediction.htm.

[28] FOG, A. How to optimize for the pentium family of microprozessors, 2000. http://www.agner.org/assem/pentopt.zip.

[29] FRANK, B., ALBERT, K., AND YAKOLEV, A. WCET analysis of superscalar processors using simulation with coloured petri nets. *Journal of Realtime Systems 18* (2000), 275–288.

[30] FÄRBER, G. *Prozeßrechentechnik. 2., völlig neubearb. Aufl.* Berlin: Springer, 1992. 223 S., 116 Abb.

[31] GRESSER, K. An event model for deadline verification of hard real–time systems. In *Proc. Fifth Euromicro Workshop on Real Time Systems* (Finland, June 1993), IEEE, pp. 118–123.

[32] GRESSER, K. *Schedulability Analysis for Event–Driven Real–Time Systems*. No. 268 in Fortschrittsberichte VDI, volume 10. VDI–Verlag, Düsseldorf, 1993. Ph.D. Thesis, Institute for Real–Time Computer Systems, Technische Universität München (in german).

[33] HARPER, J. S., KERBYSON, D. J., AND NUDD, G. R. Analytical modeling of set–associative cache behavior. *IEEE Transactions on Computers 48*, 10 (Oct. 1999), 1009–1023.

[34] HEALY, C., SJÖDIN, M., RUSTAGI, V., WHALLEY, D., AND VAN ENGELEN, R. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Realtime Systems 18* (2000), 129–156.

[35] HERGENHAN, A., AND ROSTENSTIEL, W. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings of the Design Automation and Test in Europe Conference (DATE2000)* (Paris, France, Mar. 27–30 2000), pp. 552–559.

[36] HUR, Y., BAE, Y. H., LIM, S.-S., KIM, S.-K., RHEE, B.-D., MIN, S. L., PARK, C. Y., LEE, M., SHIN, H., AND KIM, C. S. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *16th IEEE Real–Time Systems Symposium* (Pisa, Italy, Dec. 1995).

[37] INTEL. *8259A Programmable Interrupt Controller*. www.intel.com, Dec. 1988.

[38] INTEL. *8254 Programmable Interval Timer*. www.intel.com, Sept. 1993.

[39] INTEL CORPORATION. *Intel Architecture Software Developer's Manual Volume 3*. Mt. Prospect, IL, USA, 1997.

[40] INTEL CORPORATION. *Intel Architecture Optimization Reference Manual*. Mt. Prospect, IL, USA, 1999.

[41] INTEL CORPORATION. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Mt. Prospect, IL, USA, 2001.

[42] INTEL CORPORATION. *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*. Mt. Prospect, IL, USA, 2001.

[43] KAISER, A. *K7 Branch Prediction*. www.s.netic.de/ak, Dec. 1999.

[44] KENT, D. *RAM Guide*, Oct. 1998.
http://www.tomshardware.com/mainboard/98q4/981024/.

[45] KIM, S.-K., HA, R., AND MIN, S. L. 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).

[46] KIM, S.-K., MIN, S. L., AND HA, R. Efficient worst case timing analysis of data caching. In *Proceedings of the IEEE Real–Time Technology abd Applications Symposium (RTAS'96)* (1996), pp. 230–240.

[47] KLAR, R., DAUPHIN, P., HARTLEB, F., HOFMANN, R., MOHR, B., QUICK, A., AND SIEGLE, M. *Messung und Modellierung paralleler und verteilter Rechensysteme*. Teubner, Stuttgart, Germany, 1995.

[48] KOPETZ, H. *Real–Time Systems — Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[49] KUENNING, G. H. Kitrace: Precise interactive measurement of operating systems kernels. *Software: Practice and Experience 25*, 1 (Jan. 1995).

[50] LEE, C., HAHN, J., SEO, Y., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., AND KIM, C. Bounding cache-related preemption delay for real–time systems. In *18th IEEE Real–Time Systems Symposium* (San Francisco USA, Dec. 3–5 1997), IEEE, IEEE Computer Society Press.

[51] LEE, C., HAHN, J., SEO, Y., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., AND KIM, C. Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Software Engineering* (1998).

[52] LEE, C., HAHN, J., SEO, Y., MIN, S., HA, R., HONG, S., PARK, C., LEE, M., AND KIM, C. Bounding cache-related preemption delay for real–time systems. *IEEE Transactions on Software Engineering 27*, 9 (Sept. 2001), 805–826. A previous Version also appeared at the Proceedings 18th IEEE Real–Imte Systems Symposium 1997.

[53] LEE, J. K. F., AND SMITH, A. J. Branch prediction strategies and branch target buffer design. *IEEE Computer 17*, 1 (Jan. 2 1984).

[54] LI, Y., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (June 1995), ACM, pp. 456–461.

[55] LIM, S.-S., ET AL. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering 21, Nr. 7* (July 1995), 593–603.

[56] LINDGREN, M. Deriving worst-case execution time by measurements. Tech. Rep. Technical Report 00/26, Mälardalen Real-Time Research Center, Mälardalen University, Sweden, Nov. 2000.

[57] LIU, Y. A., AND GOMEZ, G. Automatic accurate time–bound analysis for high–level languages. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)* (Montreal Canada, June 19–20 1998), F. Müller, A. Bestravros, et al., Eds., Lecture Notes in Computer Science, ACM SIGPlAN, Springer–Verlag, pp. 31–40.

[58] LUNDQVIST, T., AND STENSTRÖM, P. Integrating path and timing analysis using instruction level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*

(Montreal Canada, June 19–20 1998), F. Müller, A. Bestravros, et al., Eds., Lecture Notes in Computer Science, ACM SIGPlAN, Springer–Verlag.

[59] LUNDQVIST, T., AND STENSTRÖM, P. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Realtime Systems 17*, 2/3 (Nov. 1999), 183–207.

[60] LUNDQVIST, T., AND STENSTRÖM, P. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the IEEE Real–Time Systems Symposium* (Phoenix, AZ, Dec. 1999).

[61] MATTEO, C., ROBERTO, B., AND THOMAS, G. Approximation of worstcase execution time for preemptive multitasking systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'00)* (Vancouver Canada, June 18 2000), Lecture Notes in Computer Science, Springer–Verlag.

[62] MAZZUCCO, P. *The Fundamentals Of Cache*, Oct. 2000.
http://www.systemlogic.net/articles/00/10/cache/.

[63] MÄCHTEL, M. *Entstehung von Latenzzeiten in Betriebssystemen und Methoden zur messtechnischen Erfassung*. No. 808 in Fortschrittsberichte VDI, Reihe 8. VDI–Verlag, Düsseldorf, Germany, 2001.

[64] MÜLLER, F. *Static Cache Simluation and its Applications*. PhD thesis, Deptöf Computer Science, Florida State University, June 1994.

[65] MÜLLER, F. Timing analysis for instruction caches. *Journal of Realtime Systems 18* (2000), 217–247.

[66] MÜLLER, F., AND WEGENER, J. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Proceedings of the Fourth IEEE Real-time Technology and Applications Symposium (RTAS'98)* (June 1998), IEEE, pp. 179–188.

[67] NAMARISMHAN, K., AND NILSEN, K. Portable execution time analysis for RISC processors. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real–Time Systems (LCTRS'94)* (Orlando, FL, USA, June 21 1994).

[68] PABST, T. The new athlon processor – amd is finally overtaking intel. *Tom´s Hardware Guide* (Aug. 9 1999).
http://www4.tomshardware.com/cpu/99q3/990809/index.html.

[69] PARK, C., AND SHAW, A. Experiments with a program timing tool based on source–level timing schema. *IEEE Transactions on Computers 24*, 5 (May 1991), 48–57.

[70] PETTERS, S. M., MUTH, A., KOLLOCH, T., HOPFNER, T., FISCHER, F., AND FÄRBER, G. The REAR framework for emulation and analysis of embedded hard real–time systems. *Design Automation for Embedded Systems 5*, 3 (Aug. 2000), 237–250.

[71] PUSCHNER, P., AND KOZA, C. Calculating the maximum execution time of real-time programms. *Journal of Realtime Systems* (Sept. 1989), 159–176.

[72] PUSCHNER, P., AND V. SCHEDL, A. Computing maximum task execution times — a graph–based approach. *Journal of Realtime Systems* (July 1995), 67–91.

[73] RAJKUMAR, R. *Synchronization in Real–Time Systems. A Priority Inherintance Approach*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[74] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (Sept. 1991).

[75] STAPPERT, F., AND ALTENBERND, P. Complete worst–case execution time analysis of straight–line hard real–time programs. Tech. Rep. 27/97, C–Lab, Fürstenallee 11, Paderborn, Germany, Dec. 1997.

[76] STAPPERT, F., AND ALTENBERND, P. Complete worst–case execution time analysis of straight–line hard real–time programs. *Journal of Systems Architecture, The EUROMICRO Journal 46* (2000), 339–355.

[77] STAPPERT, F., ERMEDAHL, A., AND ENGBLOHM, J. 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)* (Atlanta, Giorgia, USA, Nov. 16–17 2001), pp. 132–140.

[78] SVOBODOVA, L. *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. No. 2 in Elsevier Computer Science Library. American Elsevier Publishing Company, Inc; New York, 1976.

[79] THEILING, H., FERDINAND, C., AND WILHELM, R. Fast and precise WCET prediction by spearated cache and path analysis. *Journal of Realtime Systems 18* (2000), 157–179.

[80] US DEPARTMENT OF DEFENSE. *Military Handbook: Reliability Prediction of Electronic Equipment*. Washington DC, USA, 1991. MIL-HDBK-217F.

[81] VON BÜLOW, A. Bestimmung der WCET auf AMD–Athlon Prozessoren unter Berücksichtigung eines Realzeit–Betriebssystems, 2001. Diplomarbeit (masters thesis) at the Institute for Real–Time Computer Systems, Technische Universität München.

[82] WALL, G., BJÖRNFOT, L., AND ASPLUND, L. A source-level performance analysis tool for real-time systems. In *Proceedings of the Swedish National Conference on Real-Time Systems* (Stockholm, Sweden, Aug. 25–26 1993).

[83] WEIDERMAN, N. H., AND KAMENOFF, N. I. Hartstone uniprocessor benchmark: Definitions and experiments for real-time systems. *Journal of Realtime Systems 4* (1992), 353–382.

[84] WICKER, S. B. *Error Control Systems for Digital Communication and Storage*. Prentice–Hall, Englewood Cliffs, New Jersey, USA, 1995.

[85] WILSHIRE, P. *Real Time Linux Applications and Use*. Real Time Linux Workshop, 13-15.12.1999, TU Wien, 1999.

[86] WOLF, F., AND ERNST, R. Data flow based cache prediction using local simulation. In *Proceedings of High Level Design Validation and Test* (Berkeley, USA, Nov. 2000).

[87] WOLF, F., AND ERNST, R. Execution cost interval refinement in static software analysis. *Journal of Systems Architecture, The EUROMICRO Journal Special Issue on Modern Methods and Tools in Digital System Design* (2000).

[88] WOLF, F., AND ERNST, R. Intervals in software execution cost analysis. In *Proceedings of International Symposium on System Synthesis* (Madrid, Spain, Sept. 2000).

[89] WOLF, F., KRUSE, J., AND ERNST, R. Compact trace generation and power measurement in software emulation. In *Proceedings of International Symposium on Microelectronics and Assembly* (Singapore, Nov. 2000).

[90] WYMAN, C. *The Advanced Micro Devices Athlon Processor*. CS6810 Research Paper, 1999.
http://www.cs.utah.edu/∼wyman/classes/arch/athlon.html.

[91] YE, W., AND ERNST, R. Embedded program timing analysis based on path clustering and architecture classification. In *International Conference on Computer-Aided Design (ICCAD '97)* (San Jose, USA, 1997).

[92] YEH, T.-Y., AND PATT, Y. N. Two level adaptive training branch prediction. In *Proceedings of the 24th annual International Symposium on Microarchitecture* (Albuquerque, NM USA, Nov. 18–20 1991), ACM, pp. 51–61.

[93] YODAIKEN, V. *The RTLinux Manifesto*. Department of Computer Science, New Mexico Institute of Technology, Socorro NM 87801. www.rtlinux.org.

[94] YODAIKEN, V., AND BARABANOV, M. *RTLinux Version Two*. VJY Associates LLC, 1999. www.rtlinux.org.

Index

A Processor Description

A.1 Intel P6 Family

The Intel P6 family has been developed since 1995. The first step in this direction was the *Pentium Pro*, which was built as a replacement for the *Pentium* class processors. The roots of the P6 family go back to Intel 486, Intel 386, Intel 286 through to the 8068 processor in 1978. Various legacies in the register set, and the general structure of the processor are the result of this development.

The Pentium Pro was only produced for a very short period of time, and was soon replaced by the *Pentium II*, the *Pentium III* and recently the *Pentium 4*. The development was accompanied by minor changes in the structure (e. g. simultaneous multiprocessing *SMP* capabilities), special purpose versions (e. g. Celeron and Xeon) and resizing of the caches. The following will focus on the Pentium III as one of the newer processors. The latest available Pentium III is code-named Coppermine. Newer versions of the Pentium III, code-named Tualatin and Coppermine-T, are currently only provided to a very limited number of customers.

A.1.1 General Structure

The basic structure of the processor has not changed since the days of the Pentium Pro. Figure A.1 gives an overview of the basic building blocks of the P6 Family. One of the specialities of this structure is that the CPU behaves on the outside as a *complex instruction set computer (CISC)* but internally the core is organised as a *reduced instruction set computer (RISC)*. In the further description of the processor these RISC instructions will be called micro ops in accordance with the Intel documentation. Another notable feature is that usually the CPU core works at a higher frequency to the rest of the computer. This external frequency is called *frontside bus* or *FSB* frequency. The external frequency has increased at a much slower rate than the internal frequency and lies at present at 133 MHz. Unless otherwise noted, the CPU core frequency is of relevance which is currently pushed to a maximum of around 2 GHz.

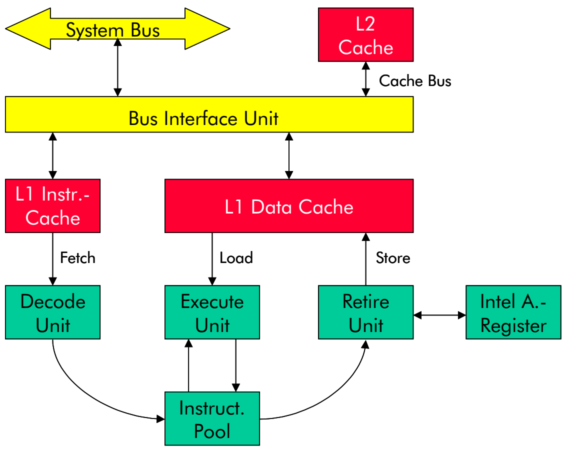


Figure A.1: Building Blocks of the Intel P6 Architecture[41]

The CPU core is made up of of 5 basic building blocks. The *fetch/decode unit* is responsible for the translation of the CISC into the micro ops and is constructed of two simple and one complex opcode decoder, which decode in parallel. Each decoder translates one CISC instruction, in one or more triadic micro ops, i. e.; two logical sources and one logical destination of each micro op. Per clock cycle, up to 6 micro ops are produced (1 for the simple and 4 for the complex decoder). A detail which has to be noted here is that, with one exception, jump and conditional jump opcodes share the complexity, i. e.; short and near jumps are simple opcodes which can be decoded in the simple decoders, whereas far jumps must be decoded by the complex decoders. The `JCXZ` opcode is the exception as regards the complexity and decoder usage. Figure A.2 provides a more detailed look into Intel's P6 Core.

The register set of the Pentium compatible processors is known to be very small. Internally, fitting for a RISC CPU, the register set is build of 40 general purpose registers. After the decoding, a dependency analysis is done to identify instructions which are independent of each other, for example, a move into a register from an arbitrary address is usually independent from previous instructions using this particular register, but not the same memory address. External registers and data are mapped to the internal register set according to their dependency on utilising the *register allocation table (RAT)*. This step is essential for the later *out–of–order* execution. The RAT is also the bottleneck of the P6 as it can issue only 3 micro ops per cycle. The fetch/decode unit itself is an *in–order* unit with speculative and parallel fetch, i. e.; the instructions are decoded in the order they are in the instruction stream. As regards the speculative part, the decode unit incorporates the branch prediction unit which is considered closely in A.1.4.

The micro ops are buffered in the *instruction pool*, formerly known as *reorder buffer*. The decode unit is only loosely coupled to the other execution units, and decodes up to

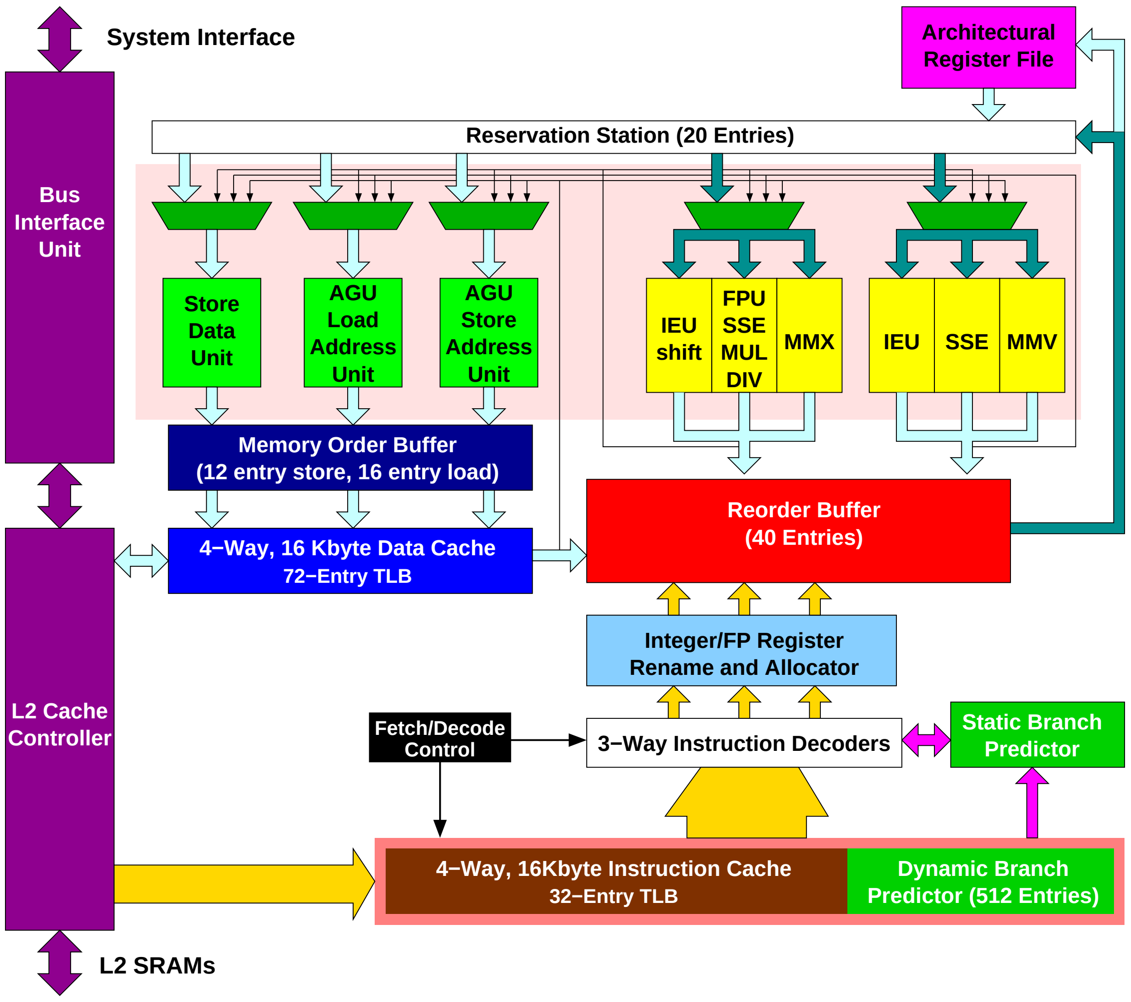

System Interface
Architectural Register File
Reservation Station (20 Entries)
Bus Interface Unit
Store Data Unit
AGU Load Address Unit
AGU Store Address Unit
IEU shift
FPU SSE MUL DIV
MMX
IEU
SSE
MMV
Memory Order Buffer (12 entry store, 16 entry load)
4–Way, 16 Kbyte Data Cache 72–Entry TLB
Reorder Buffer (40 Entries)
Integer/FP Register Rename and Allocator
L2 Cache Controller
Fetch/Decode Control
3–Way Instruction Decoders
Static Branch Predictor
4–Way, 16Kbyte Instruction Cache 32–Entry TLB
Dynamic Branch Predictor (512 Entries)
L2 SRAMs

Figure A.2: Detailed Architecture of the Intel P6 Core [68]

40 CISC instructions in advance. The instruction pool has 40 micro op registers which can each hold an micro op ready for execution, or one already executed but not yet committed to machine state.

The *dispatch/execution unit*, as an *out–of–order* unit, utilises the dependence analysis of the fetch/decode unit in order to feed the micro ops into the pipelines. The micro ops are moved into the reservation station and wait until their operands are valid and they are dispatched into a pipeline. Because of the dependency analysis this can be done while maintaining the integrity of the data operated on. The execution unit contains two floating points, two integer and one memory interface unit. The Intel documentation is inaccurate when specifying the number of micro ops possibly executed per clock. Since the memory interface can schedule 3 micro ops per cycle (1 load and 2 store), the execution units are clustered as depicted in Figure A.2 (cf. [68] and [28] Section 17). One of the integer pipelines is specially equipped for branch micro ops. It is able to detect mispredictions and, if this is the case, signals the branch target buffer to restart the pipeline. All micro ops executed speculatively are tagged if they are connected to either the taken or the not taken branch. When the branch micro op is finally executed, the pipeline signals whether the branch was taken or not, and the wrongly executed micro ops are marked as unused, thus invalidating the result. The memory interface unit is responsible for load and store micro ops. A surprising piece of information is that the opcodes for multiplication and division have a data independent execution time, i. e.; the values of the operands have no influence on the execution time (cf. [28] Section 29).

Following confirmation of speculatively executed opcodes, the results must be committed to the machine state by the *retirement unit*. The retirement unit continuously monitors the status of the micro ops in the instruction pool. Whenever a micro op is ready the result may be written to the corresponding memory, i. e.; usually the caches, or the CISC *Intel architecture register set*. After retirement of an micro op it is removed from the instruction pool. This strict separation of execution and commit of the micro ops is necessary to allow the speculative execution. Another requirement to support speculative execution is that the micro ops are retired in order, i. e.; that no micro op is retired before all micro ops preceding this in the instruction stream are retired. This is necessary to maintain data integrity in the case that the speculative execution is discarded because of a mispredicted branch.

The connection of the processor to peripheral components, and the main memory, is comprised of the *GTL+ bus* and the *north bridge*. The GTL+ bus is a shared bus, which connects all processors within a SMP system and the north bridge. One of the basic features of the GTL+ bus is that it supports multiple outstanding command/transaction pairs. The peak bandwidth is GBit/s, but this is only reached for a very short time due to communication overhead, bottleneck in the peripheral components and the main memory. As previously mentioned, the frontside bus works on a lower frequency than the CPU, which results in a certain granularity of time for all accesses to the rest of the system. A single cycle CPU delay might therefore incur no delay at all when the processor has to wait for a subsequent memory or I/O access, or it might incur a delay corresponding to the CPU frontside bus frequency ratio.

A.1.2 Memory Management

The memory of a computer system is, next to the connection to the environment, the main peripheral unit. The management of this elementary resource is therefore handled with great care. This section gives an overview of the memory organisation within the P6 family. Caches, as a matter of processing acceleration, are discussed separately in the next section.

The memory of the P6 family can be accessed in three ways:

1. Flat model.
2. Segmented model.
3. Real–Address Mode model.

In the flat model, the memory is addressable from 0 to 4 GByte. Code, data and I/O are all within the same linear address space. An address for any byte, within this address space, is called a *linear address*. In the segmented model the *logical address* is comprised of a segment selector, and an offset within this segment. Figure A.3 depicts the translation scheme. 8192 Segments of different types and sizes, with each segment up to 4 GByte addressable space, are supported by the Intel architecture. Flat model and segmented model are referred to as protected mode.

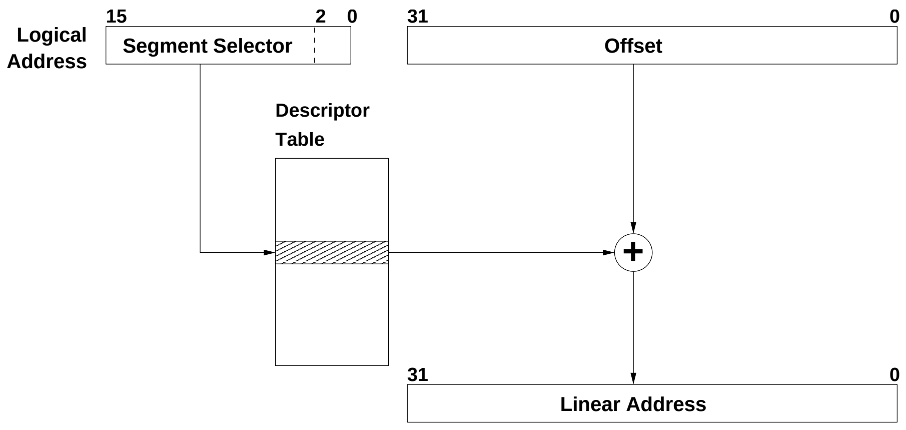


Figure A.3: Translation of Logical to Linear Addresses [81]

Bit 0 and 1 in the Segment selector specify privilege level which is likely to be the highest priority in the case of real time systems. Bit 2 indicates if the *Local* or *Global Descriptor Table (LDT* and *GDT)* should be used. The remaining bits, from 3 to 15, form the index which selects the segment descriptor in the GDT or LDT. Sections 3.4 and 3.5 in [39] give in–depth information on the subject of segment selectors, and the corresponding descriptor tables.

The real–address mode model dates back to the times of 8086. It is composed like the segmented model of segment selector and offset, but the segment size is fixed equally to 64 KByte, and the maximum linear address space of 1 MByte.

The primary reason for using the segmented model is the protection the different segments provide. With this protection, it is possible to prevent, for example, the stack from growing into the data or code. Because of the irrelevance of real address mode and flat model, further discussion will concentrate on the segmented model. A additional abstraction layer is introduced by the *paging mechanism*, which is transparent to the application programs and generates additional flexibility and overhead for the system as a whole. With the paging mechanism, the available memory is organised in page frames of fixed size, which are usually aligned to a multiple of the page size. The page size is the quantum of memory which can be alloted to a task and is typically 4 KByte. Sometimes it might be reasonable to use larger page sizes or to mix different page sizes, but generally only 4 KByte pages are utilised.

In order for the paging mechanism to work it has to be supported by the operating system and the hardware. Up until now the discussion was limited to the computation of the linear from the logical, which are often referred to as *virtual addresses*. Now the mapping from a linear to a *physical address* is considered. Figure A.4 shows how the mapping from linear to physical address is accomplished.

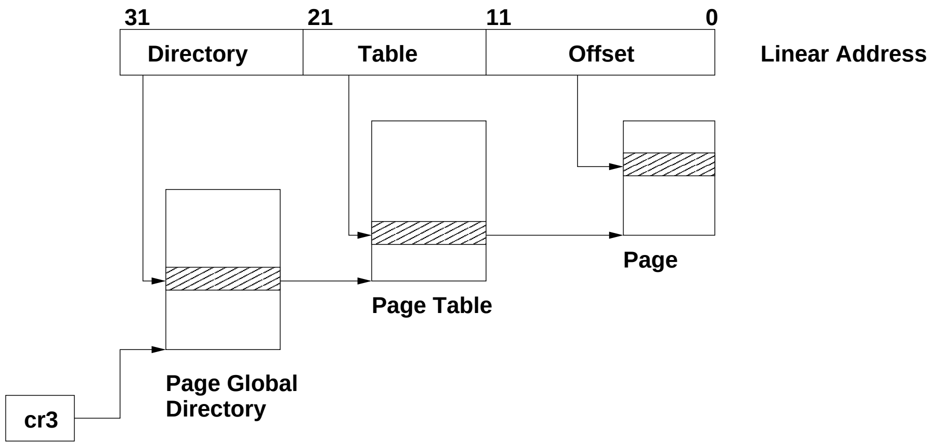


Figure A.4: Mapping of linear to physical addresses [81]

The linear address is split up into directory, table and offset part. Register *cr3* points to the *page global directory*, where the directory portion of the address selects an entry pointing to a page table. Within the page table, an entry is chosen by the table part of the address. This entry points to a physical page, and the offset portion of the linear address points to the final physical address within this page. The mapping is done within the processor hardware, but the page directory and page table reside in the main memory and are usually administered by the operating system.

The paging mechanism allows the applications and the operating system to operate on fixed virtual address space, even if the physical addresses might change from one execution to the next, or despite swapping of memory portions to external storage devices. Both changes are undesirable, and stem from the dynamic memory management used in general purpose systems. The information regarding the paging mechanism is essential for the test setup in Section 5.3. See [39] Sections 3.6 and 3.8 for further information on the subject of paging.

The *translation lookaside buffers (TLB)* are little caches for the paging mechanism. As previously explained, the paging mechanism is based on two kind of tables residing in main memory: the page global directory and the page tables. In order to read a value from main memory three memory accesses are needed to retrieve first the page global directory entry, then the page table entry and finally the desired data. Because the main memory is one of the slowest components in the computer system, a workaround is used. Most processors providing a paging mechanism also provide caches for the reference information. The Intel P6 family processors provide TLB entries for the most recent pages, seperately accessed for instruction and data. The page sizes supported are; 4 KByte, 2 MByte and 4 MByte. The number of entries provided varies from type to type but can be read with the `CPUID` opcode described in the following section. The usual configuration has 32 instruction TLB entries, and 64 data TLB entries for 4 KByte pages. It might happen, due to TLB misses, that two main memory accesses for the page global directory, and the page table entry are necessary for the hardware to register that it has an L2 Cache hit.

A.1.3 Caches

One of the main problems in modern microprocessors is keeping the instructions and data flowing into the processing unit, at a rate that the processor has not to wait too long for the memory fetches. The main memory speed has not developed as quickly as the processor. Since the introduction of the Intel 486 DX, caches for instructions and data were added. Older generations included only special purpose caches for the TLBs (cf. Section A.1.2) and the *segment registers*. This section will solely concentrate on caches for data and instructions. The cache is, next to the frequency development of the CPU, the main reason for the high computing power of modern computers.

The cache structure of the P6 family is organised at two levels. While the first level *(L1)* is split up into instruction cache and data cache, the second level *(L2)* cache is organised as a uniform cache, i. e.; the cache holds data as well as instructions which were previously used. The P6 family utilises an inclusive cache design i. e.; data and instruction in the first level cache are also contained in the second level cache. In contrast, the AMD Athlon has an exclusive cache design, i. e.; the data and instruction in L1 are not duplicated in the L2 cache (see Section A.2.3). The size of a cacheline is generally 32 bytes. It should be noted, that the physical address in main memory is the deciding factor for which set in the caches is chosen for the memory location storage. The logical and linear address have no impact on this.

Due to the fact that the Pentium II and III share the main features as regards the caches whereas the Pentium Pro differs considerably, the Pentium Pro will not be considered further in this discussion.

The L1 cache of Pentium II and III is placed on a chip. There are 16 KByte each for instructions and data. While the documentation (cf. [40]) states that both instruction and data cache are 4-way set associative, tests with the `CPUID` opcode suggest that, on some processors types, the data cache is 2–way set associative. The `CPUID` opcode provides information on the processor type, the model specific extensions and the cache sizes (see pages 3–114 to 3–127 in [42]). The cache associativity relays the number of cachelines a physical memory location can be mapped to. *pseudo least recently used (pseudo–LRU)* is utilised as replacement scheme for the L1-Cache . The documentation (cf. [40]) neither states the implication of the “pseudo” prefix, nor the replacement scheme of the second level cache.

The L2 cache is located off chip, but on the processor module, and is 128 KByte or more in size. The most recent Pentium III, the Coppermine, has 256 KByte while older Versions (Katmai) had 512 KByte. While the cache size has been cut down, the speed of the L2 has been doubled. The Katmai worked with halve processor speed L2, and full processor speed for L1 cache, while the Coppermine uses full processor speed for both caches (cf. [28] Section 31). Further discussion will focus on the full speed L2 cache. The second level cache is 4-way set associative. The timing data provided here is taken from Intel’s optimisation guide [40] Section 1. These values can be used as a guideline and plausibility check, but must be verified on the given system utilising measurements. A Cache miss in L1 cache, with a hit in L2 cache, induces 4 to 10 clock cycles. An L2 miss needs at least 11 FSB cycles when the corresponding entry in the TLB , is present (page table hit) and 3 additional FSB cycles on a page table miss. These cache load times have been confirmed by measurements (cf. Section 5.3.2). As replacement scheme for the L2 cache LRU is used as with the L1 cache.

The processor supports several methods for the caching strategy. These are not only set for the whole system (register CR0), but are tagged to the page table and directory entries as well, and can be set in the *memory type range register (MTRR)* for memory regions. Though memory is the main application field of the caches, memory mapped I/O is handled in the same way. This explains the number of caching strategies supported which are briefly explained in the following.

Uncacheable (UC) These memory locations are directly read and written over the system bus. No caching mechanisms apply.

Write Combining (WC) Like in UC, no caching is used, but the write accesses may be delayed and combined in the write buffer. This reduces memory accesses over the system bus considerably.

Write Through (WT) In this case, read accesses go into the caches, but whenever a write is issued by the processor the data is written not only to the cacheline but also back to memory, with only a short possible delay in the write buffer.

Write Back (WB) If write back mechanism is enabled, reads and writes to the memory are cached. Whenever cachelines are deallocated in the L2 cache, and a cacheline is dirty, i. e.; a write operation was performed on the cacheline, the cacheline will be written back to main memory. This is the most frequently used caching strategy, since it avoids a lot of unnecessary writes to memory.

Write Protected (WP) Write protected is very similar to WT, but write commands are written to the memory locations and the corresponding cacheline is invalidated.

More details on the strategies are described in Section 9.3 in [39]. It has to be noted that, for the monitor, the write back strategy must be taken into account.

As a small side effect, it has to be considered what happens when a cacheline resides in L1 data and L1 instruction cache. This can happen when code and data are not properly separated, or in the case of self modifying code. Both situations are absolute malpractice, even in non real-time applications. When a cacheline in data cache is written, which is also held in the instruction cache, the instruction cacheline is invalidated and reloaded on the next request. If the instruction executed on the modified cacheline follows closely to the modifying instruction, a complete processor stall is initiated, i. e.; all computed results of the instruction in the modified cacheline will be invalidated, and after the cacheline fill in the instruction cache the decoding and processing of the instruction will be restarted. Obviously this has to be avoided and is fortunately taken care of already by means of compilers and linkers. Thus, only hand optimised assembler code has to be checked for this situation.

A.1.4 Branch Prediction

In addition to the caches, a number of other optimisation schemes are implemented in the P6 family processors. They either belong to the category of branch prediction and speculative execution, or they optimise the segmented memory accesses and paging.

The documentation from Intel ([39] Section 13.2.1, and [40] Section 2) is not completely consistent as regards the branch prediction. Based on the sparse information from Intel, papers from Agner Fog ([27] and [28]) and my own tests on the hardware, the following branch prediction model has been chosen and utilised.

The branch prediction scheme is closely coupled with the fetch/decode unit. A branch is defined as a possible load into the *instruction pointer* register. Such a load occurs whenever the next instruction to be executed is not the succeeding instruction in memory. The branch opcodes are `CALL`, `RET`, and either conditional or unconditional jump (`JMP` and `Jxx`) opcodes. The interrupt, and return from interrupt, (`INT` and `RTI`) opcodes can be considered as branch opcodes, but are not covered by the branch prediction mechanism.

Whenever the fetch/decode unit encounters a branch opcode, the first test is whether the address of the instruction is in the *branch target buffer (BTB)*. The BTB of the P6 family has 512 entries organised 16 way set associative, and the replacement scheme utilised is pseudo random (cf. [28] Section 22). Again the term *pseudo* is not explicitly specified, but can be expected to indicate a dependency on time and/or history. Bits 4 to 31 of the address of the last byte of the branch instruction are used to identify the branch entries. Bits 4 to 8 define the set, while all bits are used as the tag which is stored in the BTB.

When a valid entry for the branch is found in the BTB the dynamic and otherwise static branch prediction scheme is used as depicted in Figure A.5. In the case of unconditional branches, the branch is reasonably predicted as *taken*.

If the target address is unknown, the conditional branch is assumed to be *not taken*, and the subsequent instruction is fetched. This is called *fall through*. This behaviour is reasonable, since when the target address is unknown the other possibility for the decode unit would be to be idle until the target address is known, which is usually not earlier than the result if the instruction will branch. Because of this, the penalty for a *misprediction* is small compared to no prediction at all. A source when this happens are, for example, switch statements in C which are translated into indirect or indexed branches. With these it has to be considered that an register indirect or indexed branch is assumed to branch to the same address it did last time this instruction was encountered. If the target address is known, the branch is assumed *taken* when the branch is directed backward, and assumed *not taken* when the branch is directed forward. A backward branch is usually a loop, where the prediction *taken* will lead to sensible results. In the case of forward branches, following the branch would incur an additional cycle. In the case of unconditional branches, these are predicted *taken*, even if the target address is unknown. Since the address may be provided slightly before the instruction is completely through the pipeline, up to six cycles are lost (cf. [40]).

The dynamic branch prediction scheme is described for Intel Pentium class, and P6 family processors in [27]. The dynamic prediction is divided into two levels. The first level was developed by Lee and Smith [53], and is called *smith prediction scheme*. It is already implemented in the Pentium class processors, and consists of a two bit *history counter* with saturation, i. e.; the counter will not wrap around at decrementing 0 or incrementing 3.

Figure A.6 shows the finite state machine of the prediction. The static transition is

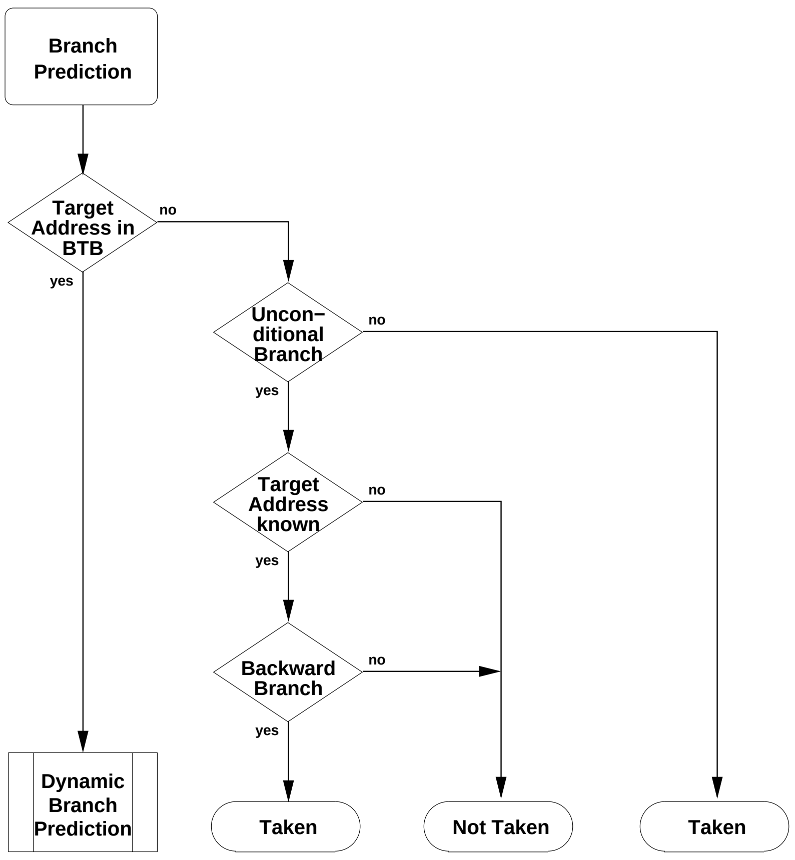


Figure A.5: Static Branch Prediction Algorithm in Intel P6 Family

marked with a + correspond to a *taken* branch, and the ones marked with a - correspond to a *not taken* branch. If the counter is in the lower two states (0 and 1) the branch is predicted to be *not taken*, while in the upper states (2 and 3) the prediction is *taken*. In the states 0 and 3 (strong *not taken*/*taken*), at least 2 consecutive occurrences of this instruction must deviate from the recent behaviour in order to change the prediction. This avoids infrequent different behaviour (e. g. the controlling branch instruction of a nested loop leaving the previous execution of the loop) from interfering with the prediction scheme.

As second level of the dynamic branch prediction scheme, the work of Yeh and Patt [92] has been used. Each BTB entry contains 16 of the previously described history counters, and a 4 bit history shift register. As depicted in Figure A.7, the history shift register selects one of the 16 history counters to be used for the branch prediction. Whenever a branch is actually taken, a 1 is shifted into the history, otherwise a 0 is inserted. In this way, repetitive patterns with a period of up to 16 are detected, and after a short learning phase correctly predicted. Agner Fog describes detailed detectable patterns in

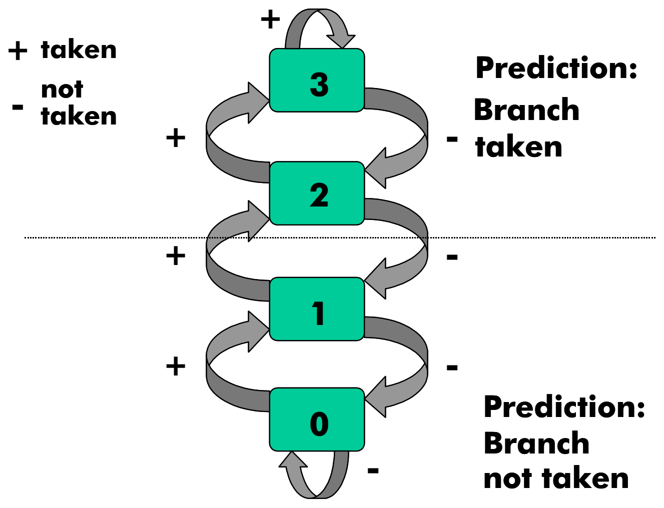


Figure A.6: First Level of the Dynamic Branch Prediction of the P6 Family[27]

Section 22.2.4, in [28]. Periodic patterns with a period of up to 5 are predicted correctly, after no more than two periods.

A further step in the branch prediction is the *return stack buffer (RSB)*. This entity stores the return address of, up to 16, calling hierarchies. Thus whenever a return opcode is met, and the RSB is not empty, it will start decoding rather than waiting for the stack to yield the address of the next instruction. Using the assumption that calls and returns are always paired, the performance can be considerably increased. If the RSB is empty the default branch prediction scheme described previously is used, i. e.; the `RET` opcode is expected to branch to the address it has previously returned. In order for this to work, the return utilises a BTB entry, regardless of whether the RSB is full or not. When a nesting level deeper than 16 is reached, the innermost 16 entries are provided by the RSB, while the outermost entries have to rely on a BTB entry, or as a last resort the common stack (cf. Section 22.2.11 in [28]).

A.1.5 Builtin Monitoring Support

The Intel P6 family provides six debug registers to allow for generation of a debug exception on up to four addresses. The debug registers can be tuned to trigger an exception on either the execution of the opcode, data write or read access, or I/O read or write access at the given address.

In addition, the processor family has a number of model specific features that can be utilised for the monitoring purpose. One is the *time stamp counter (TSC)*, which counts

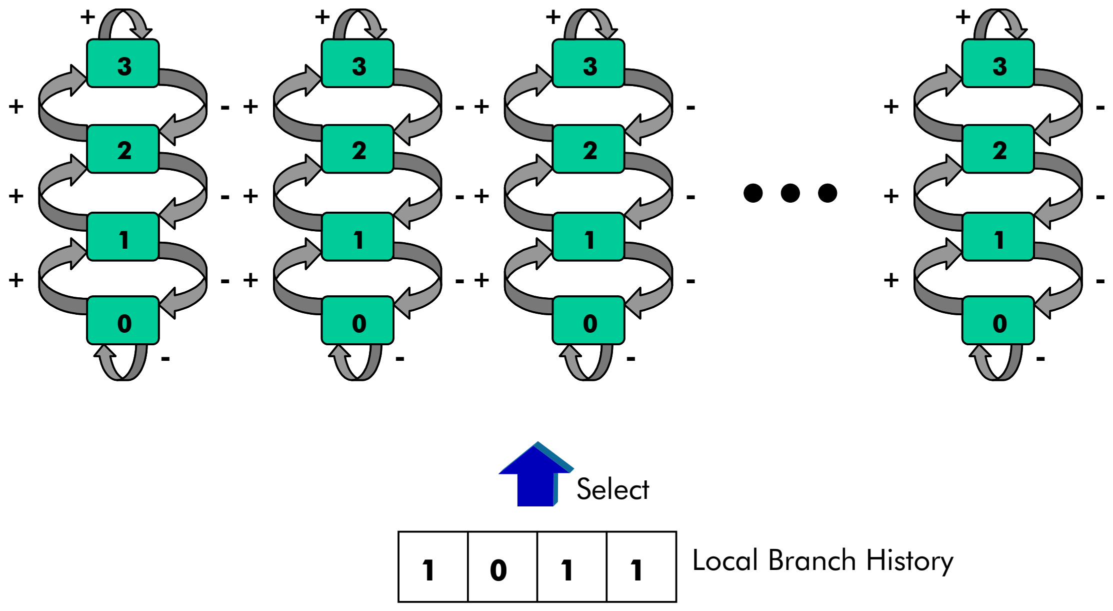


Figure A.7: Second Level of the Dynamic Branch Prediction of the P6 Family[27]

the clock cycles in a 64 bit counter from power on (cf. Section 14.5 in [39]). This allows one to perceive the measurement of code with a resolution in CPU cycle, without the additional overhead of interrupt programming. There are a good many other model specific registers which go well beyond the scope of this work and these can be found in Section 8.4 in [39].

The two *performance monitoring counters (PMCs)* allow the recording of a number of internal events, or the duration a certain condition is true. Internal events that can be monitored with these counter are cache misses (individual for each cache), TLB misses and branch prediction errors. Further detail on other possible events can be seen in Appendix A in [39].

A.2 AMD Athlon

The AMD Athlon processor family resembles the Intel P6 family considerably. Much of this similarity stems from the fact that AMD is *"only"* second source, and must therefore be compatible with Intel's x86 processor families. A short tabular comparison between AMD Athlon and Intel Pentium III, from AMD´s point of view, is given on page 3 in [3]. The term *Athlon* actually refers to the whole of the AMD Athlon family up to the 3rd generation of Athlon processors, which is code-named *Thunderbird*. The 4th generation, code-named Palomino is not considered throughout this dissertation.

The following sections will focus on the similarities and differences between the AMD Athlon and the Intel Pentium III described in previous sections.

A.2.1 General Structure

The basic structure, as depicted in Figure A.1 for the Intel architecture, is also valid for Athlon, while in detail varying considerably. As with the P6, the Athlon is externally a CISC machine and internally has a RISC like load/store architecture. It is also similiar to the P6 super scalar, i. e.; fed with fitting instructions, it can execute up to three instructions per clock. One of the basic requirements to support this is the *3-way instruction decoder*. Unlike the P6 family, the 3 decoders inside the unit are identical, i. e.; each instruction can be fed to any of the three decoders.

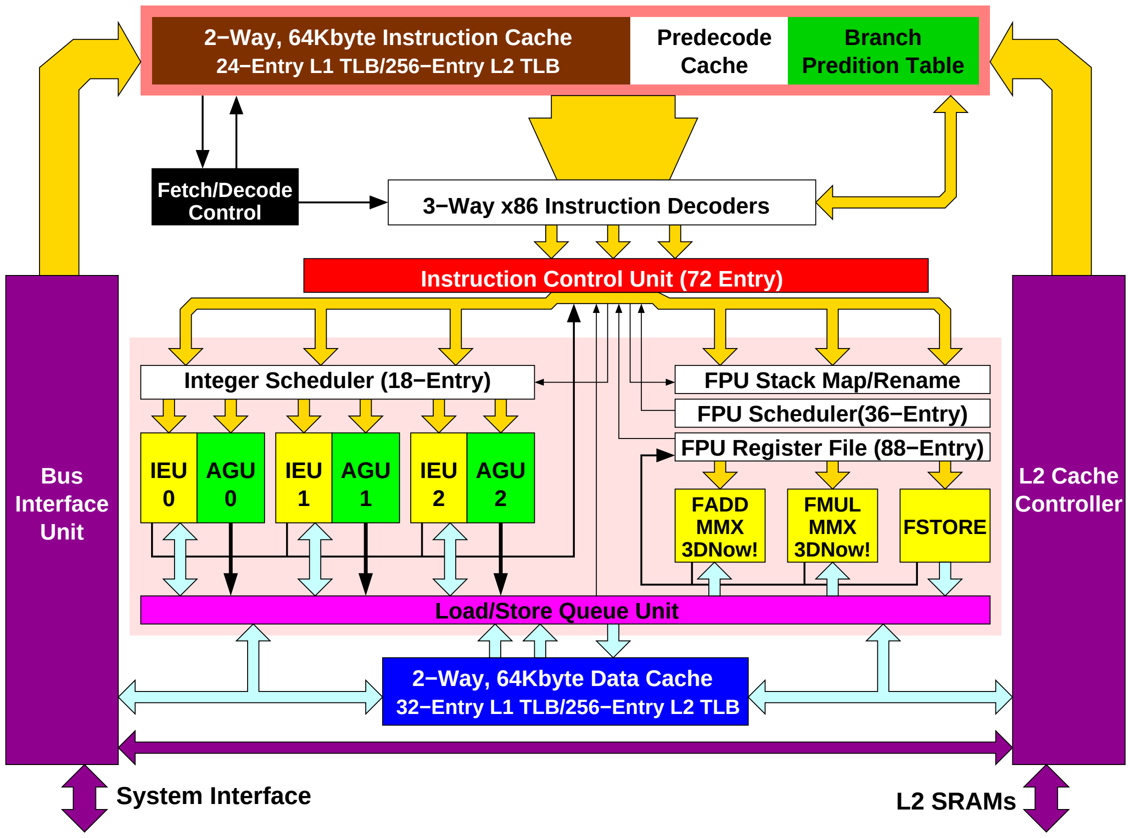


Figure A.8: Basic Architecture of the AMD Athlon Core [4]

The processing core is built up of 9 parallel execution pipelines: 3 *integer execution* *units (IEU)*, 3 identical *address generation units (AGU)* and 3 *floating point units (FPU)*. The IEUs are identical, except for the *integer multiplication unit (IMUL)* attached to the IEU0 and IEU1. The question arises, why implement 9 execution pipelines when only a three-way decoder is used, and there is usually only enough instruction level parallelism to feed a 4-way super scalar processor?

The general design outline in many RISC machines is aimed at keeping the common case fast, and the uncommon case correct. This has led in the Athlon to a dual implementation of the decoding unit as regards instructions. There is a *direct path* for the usual instructions, which are decoded into one or two *macro ops (MOP)*. Unfortunately the documents do not state, in how many MOPs an individual instruction is decoded. The MOPs are register to register operations and therefore, while it is only implicitly hinted by the term “load/store architecture” in the AMD documentation, a mapping of operands to the internal register set is done (cf. [90]). They are similiar to Intel’s micro ops RISC commands, but with a fixed 64 bit of length. This leads to additional MOP level parallelism. The decoding unit can issue up to three MOPs per cycle. In the case of uncommon instructions the *vector path* is used. The vector path utilises ROM routines to decode these instructions rather then decoding in hardware. Per cycle only one complex instruction can be fed into the *microcode engine*. Appendix G in [4] defines, for all instructions, which of these instructions are decoded in the direct path and which in the vector path. Unfortunately, the rule implied at the start of the section has been somewhat degraded for the Athlon. Important, and probably very common, opcodes like `POP`, `OUT`, `MUL`, `IMUL`, `DIV` and `IDIV` are implemented in the vector path. The MOPs produced with the microcode engine are dispatched, together with the MOPs from the simple decoders to the *instruction control unit (ICU)* which in turn feeds these either to the FPU or integer scheduler. The ICU, the integer and FPU scheduler are the core of the out-of-order execution of the AMD Athlon.

The x86 floating point operations work with the floating point stack. This is a drawback for a fast floating point processor. In order to circumvent this problem a remapping of this stack to the 88-way internal register set is done. This is the first step in the floating point pipeline after the decoding. The FPU scheduler is responsible for issuing the floating point MOPs to the execution units. The three units are differently organised. The first floating point unit is known as the *adder pipe* and supports 3DNow! add, MMX ALU/shifter and simple floating point arithmetic. The second unit is the *multiplier pipe*. It contains, in addition to 3DNow!/MMX multiplier and reciprocal unit, an MMX ALU and a floating point multiplier/divider/square root unit. The last is responsible for loading and storing operands. This step must be done separately from the load/store queue unit, since the mapping of floating point stack to the internal register file must be taken into account. Additionally, many MOP vector paths are mapped to this pipeline.

The EV6 bus, used to connect the processor with the host bridge, was developed for the Alpha 21264 Processor, by the Digital Equipment Corporation (cf. [90]). In contrast to

Intel’s shared GTL+ bus, the EV6 has a point-to-point topology. In this way, the impact of other processors, within an SMP machine is reduced. To protect the data and instruction transfer additional ECC data is transmitted. As with the GTL+, it is a packed based bus which allows 24 outstanding transactions, but utilises source synchronous instead of common clocking, i. e.; the sender of data provides the clock signal. Thus, the delay on the bus is negligible. This is one of the main reasons why the bus running currently and effectively on 200 MHz – actual clocking is 100 MHz, but the transmission is done on the rising and the falling edge – is scalable up to 400 MHz and beyond. Current peak transfer is 1.6 GBit/s. As with the Intel P6 family, the Athlon processor is clocked with a frequency which is higher than the rest of the system. Therefore, the Athlon suffers the same quantisation of execution time delays, due to the access of processor external resources.

A.2.2 Memory Management

Athlon memory management is very similar to that of Intel. A major difference would jeopardise the compatibility. The Athlon has only been tuned with regards to the TLB entries. The documentation (cf. [4] Appendix A) states the following data.

The TLB is split for L1 cache, L2 cache and instruction and data. The first level *data TLB (DTLB)* is fully associative, and contains 32 entries (24 for 4 KByte pages and 8 for 2 MByte and 4 MByte pages), while the second level DTLB only supports 4 KByte pages with 256 entries, and is 4-way set associative. The *instruction TLB (ITLB)* for L1 cache is also fully associative, and contains 24 entries (16 for 4 KByte pages, and 8 for 2 MByte and 4 MByte pages). The L2 ITLB is 4-way set associative, and contains 256 entries which can only map 4 KByte pages.

A.2.3 Caches

AMD Athlon cache design is very similar to that of the Digital Alpha 21264 (EV6) (cf. [62]). The repeated use of Digital Equipment Corporation techniques, which have been licensed by AMD, can be explained by the fact that the development of the Athlon was led by Dirk Meyer, who was also head of development of the DEC 21264 at the Digital Equipment Corporation labs.

Again, like the Intel P6 family, the Athlon has a two level cache design. The first level is split into data and instruction cache, each 64 KByte. AMD started to produce Athlons with 512 KByte L2 Cache on module. The tags for these were full speed on chip to allow for a fast miss detection. Later they moved the L2 on chip, and reduced it to 256 KByte. This step is consistent with Intel’s decision to halve the cache from Pentium III

Katmai, to Pentium III Coppermine. While the external L2 Cache could be clocked with 1/3, 2/5, 1/2, 2/3 or 1 times the processor frequency, with a recommended value of 1/3, the newer on–chip L2 is clocked with processor frequency. The move of the cache was possible, due to the further miniaturisation from 0.25 micron to 0.18 micron technology, and nearly doubled the number of transistors from about 22 millions to approxamiately 37 millions (cf. [90] and [5]).

A cacheline in AMD Athlon consists of 64 byte as compared to Intel's 32 byte. A speciality of the L1 data cache , mentioned in the AMD documentation (cf. [5] and [3]), is its organisation into 8 banks thus allowing 2 concurrent load/store accesses of 64 bit. Instruction and data cache are dual ported, with dedicated snoop tags to avoid the system coherency traffic. Predecoding begins when the instruction cache is filled. At the load, additional information is generated, to help the efficient detection of boundaries between the variable length x86 instructions, to distinguish direct path and vector path instructions, and to identify branch instructions and their type. This information is stored alongside the instruction cache.

Another feature of the Athlon L1 instruction cache is the implicit *prefetching*, aside from the MMX `PREFETCH` opcode. Whenever a instruction cache miss occurs, not only the cacheline containing the instruction is loaded, but also the subsequent cacheline, causing the hit rate of the instruction cache to increase.

The L1 caches are each two way set associative, while the L2 is 16–way set associative. Higher associativity has the advantage of higher hit rates, but at the cost of higher latencies due to the search, and the need for a serious amount of additional transistors. To provide optimal average performance, the L1 cache is kept fast by keeping associativity low, and the L2 is designed with higher associativity to keep the hit rates high.

In contrast to the Intel P6 family the AMD cache design is exclusive, i. e.; the data residing in the L1 cache is not duplicated in the L2 cache. Whenever data is displaced out of the L1 cache, it has to be moved into the L2 cache. In order to accelerate the move to the L2 in exclusive cache designs, the *victim buffer* or *victim cache* is introduced (cf. [62]). In the case of the Athlon, the victim buffer can hold up to 8 cachelines. The transfer to the victim buffer is done parallel to the load of the new cacheline into L1. If the victim buffer is full, the write back is triggered and all the 8 Cachelines are written back to L2. Figure A.9 shows the worst case effect in clock cycles of such a full victim buffer write back, i. e.; 8 cachelines. First of all, the data is transferred into the L2 Cache, then the L2 needs two additional cycles before providing the data. In the last step the L2 needs a further two cycles before being ready for new commands. In the case of a L2 miss, the fetch time for the relevant data will usually exceed the write back of the victim buffer.

In order to save time, the victim buffer is drained, i. e.; partially written back whenever the L2 cache interface is idle. Whenever a L1 Cache miss occurs, the victim buffer is

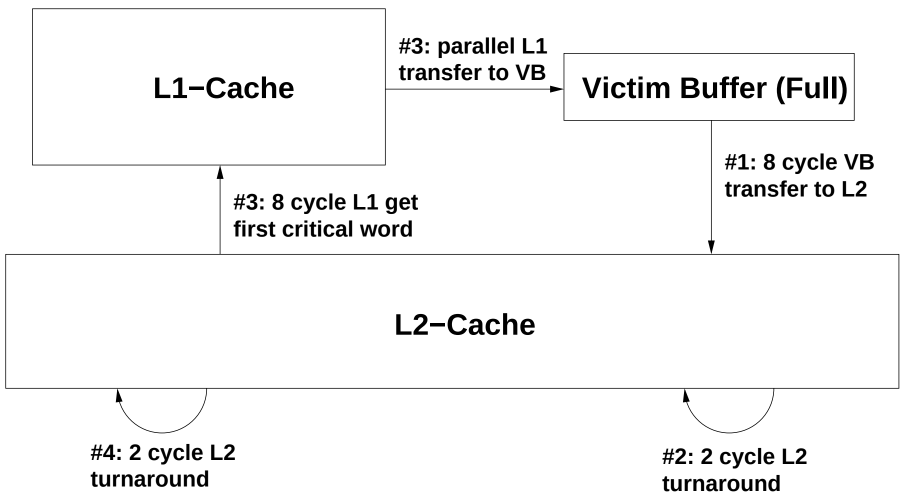

L1−Cache
#3: parallel L1 transfer to VB
Victim Buffer (Full)
#1: 8 cycle VB transfer to L2
#3: 8 cycle L1 get first critical word
L2−Cache
#4: 2 cycle L2 turnaround
#2: 2 cycle L2 turnaround

Figure A.9: Victim Buffer within an L1/L2 exclusive Cache Design [5]

searched first. If the required data is found in the victim buffer, the content of the victim buffer has to be stored in the L2 and then the data is fetched back into L1. This is due to the one way ports of the victim buffer, which saves a considerable amount of transistors.

A.2.4 Branch Prediction

The branch prediction of the AMD Athlon family is more complex by far when compared to that of the Intel P6 family. It consists of four parts:

- Predecode Cache
- Global History Bimodal Counters (GHBC)
- Global Branch History (GBH)
- Branch Target Address Cache (BTAC)

A few decisions are taken in the predecode step. These are similar to the static prediction scheme of Intel.

One major difference to the *“static”* branch prediction of the P6 family, is the distinction between short branches on one side, and near and far branches on the other. Short branches come with a single byte offset as immediate operand, only allowing for 127 byte forward or 128 byte backward branch, relative to the consecutive instruction of the branch instruction. The branch target address of these instructions is computed on

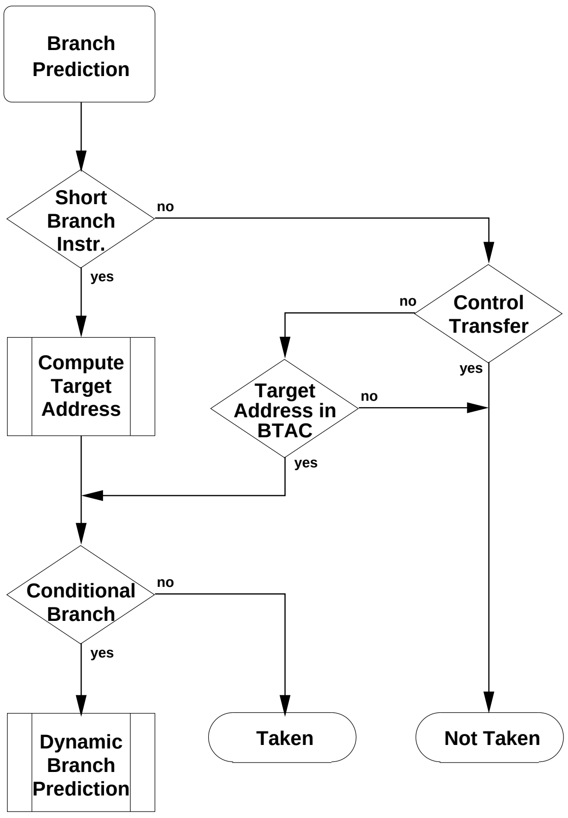


Figure A.10: Prediction Scheme of the AMD Athlon

the fly, without utilising the AGU of the processor core. The difference between near and far branches is that a near branch stays inside a memory segment, and a far branch enters a new segment. Far branches can be further distinguished into control and non–control transfers. A control–transfer changes the privilege level, i. e.; system call or task change in general purpose systems. In this case the branch is always *“mispredicted”*. Non-control transfers stay within the same privilege level. With near and non-control far branches, BTAC is tested for a valid entry and, if this test fails, the branch is predicted to fall through. In the case of unconditional branches, the branch is generally predicted *taken*, with the exception of an invalid BTAC entry. If that is the case, even unconditional branches are predicted *not taken*. This does not slow down performance, due to the unknown target address. Conditional branches with known target (i. e. BTAC hit or short branch) utilise the dynamic branch prediction mechanism described below.

The predecode cache is closely coupled with the instruction cache. An instruction cacheline is partitioned in four blocks, of 16 bytes each. Each block is assigned to a predecode cache entry. While each predecode cache entry is assigned to two blocks, the blocks do not stem from the two corresponding cacheline blocks within a cache set, but from two different sets. Figure A.11 shows the assignment. This fact would make the modelling even more complex, since the two entries within a cache set are not equivalent, i. e.; the behaviour will most likely differ, if the entries within a set are swapped.

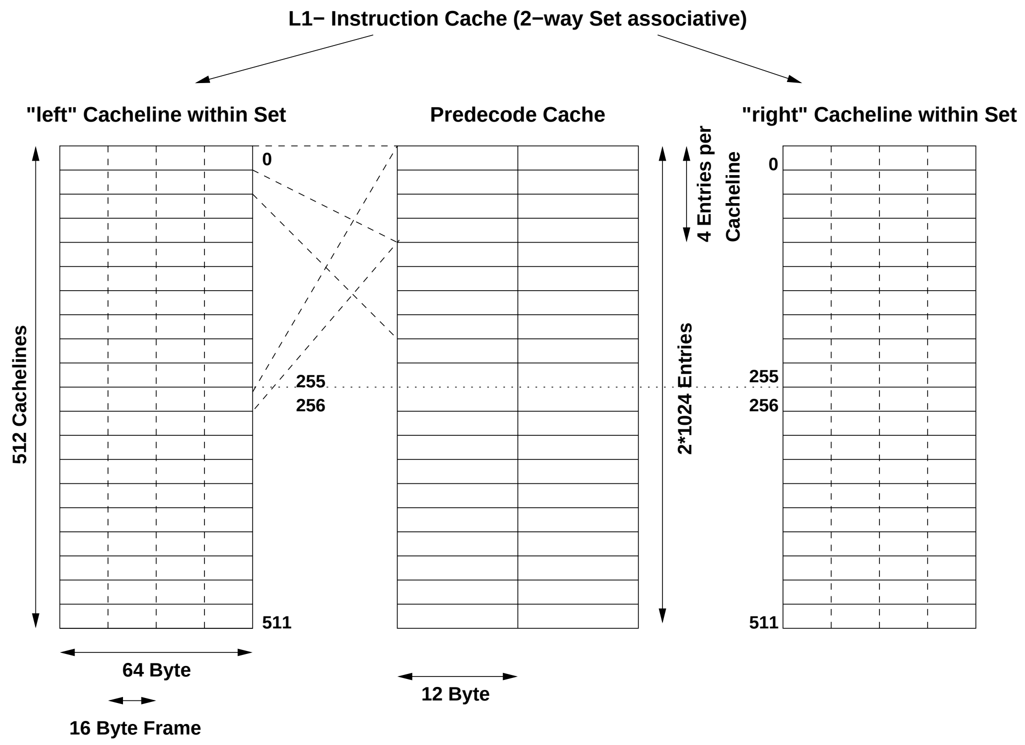


Figure A.11: Assignment of Cachelines to Predecode Cache Entries [81]

Each predecode cache entry provides three blocks for the branch prediction: Two for branch instructions and one for a return. Thus *“only”* two branches, and one return instruction within an aligned 16 byte frame, can be handled by the prediction scheme. Instructions exceeding this limit are predicted to fall through. Since a jump utilises at least two bytes, the possibility for more than two branch instructions within a frame is rather low.

The prediction as to whether a branch is *taken* or *not taken*, is done by the *global history bimodal counters (GHBCs)*. They are built equivalent to Intel’s first level branch prediction scheme, as described in Figure A.6 in Section A.1.4. There are currently 2048 GHBC implemented in the AMD Athlon family. The predecessor, the AMD K6-2 had designed 8096 into them. While strongly reducing the number of GHBCs, the branch prediction accuracy has not degraded from K6–2 to the Athlon. AMD claims a prediction accuracy of 95 % which is slightly higher than Intel’s 90 %. This is due to the organisation in 256 columns and 8 rows, and the second level of the prediction scheme explained in the following paragraph.

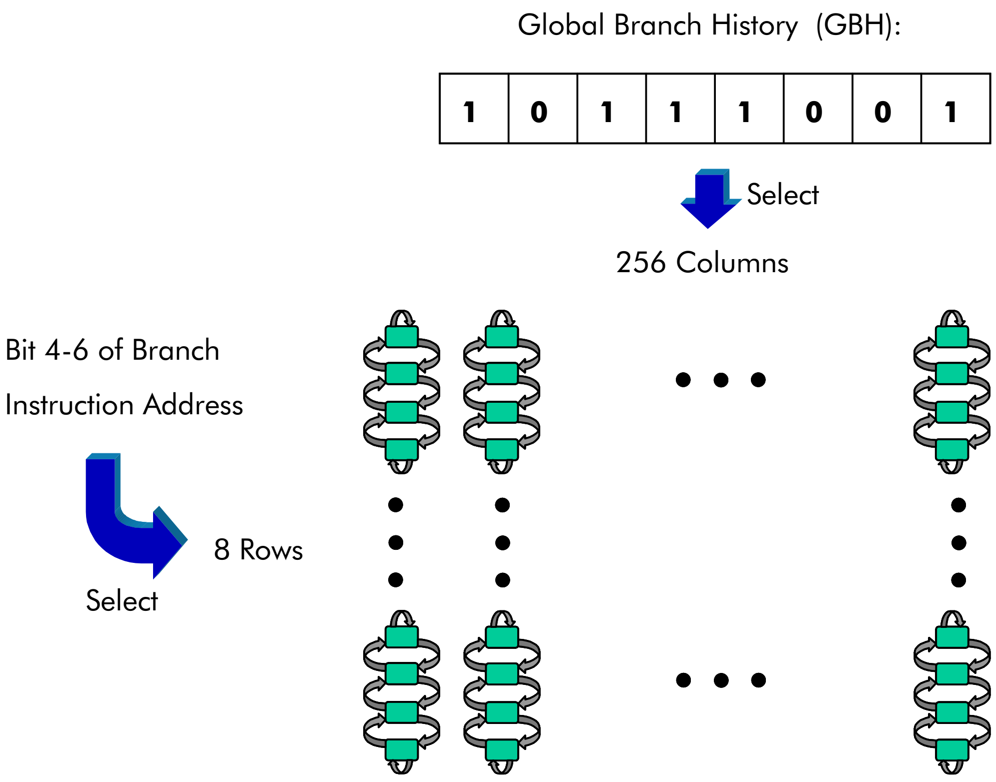


Figure A.12: Overview of the Dynamic Branch Prediction of the AMD Athlon

The second level of the prediction scheme is, at first sight, very similar to Intel's. The global branch history is an 8 bit shift register which stores whether the branch was taken or not, individually for the last 8 retired branches. The GBH addresses the column of the GHBC. The rows are selected by bits 4 to 6 of the branch instruction address, i. e.; a predecode cache entry. Thus an individual branch instruction can use up to 256 GHBCs, but each counter can be potentially used by 256 predecode entries. The algorithm is based on the fact, that in reality only a small set of GHBCs are used by an individual instruction. By not allocating individual GHBCs to a predecode cache entry, the utilisation of the available counters is increased. In order to support this feature, the GHBCs are never reset.

Corresponding to Intel's BTB scheme, the target address of a branch instruction is stored in *branch target address caches (BTAC)*. The AMD Athlon has 2048 BTAC entries, which correspond to the predecode entries as depicted in Figure A.13.

As each predecode cache entry supports up to two branch instructions, two predecode cache entries share one *branch target address cache (BTAC)* entry. Within the Athlon 2048 BTAC entries are provided.

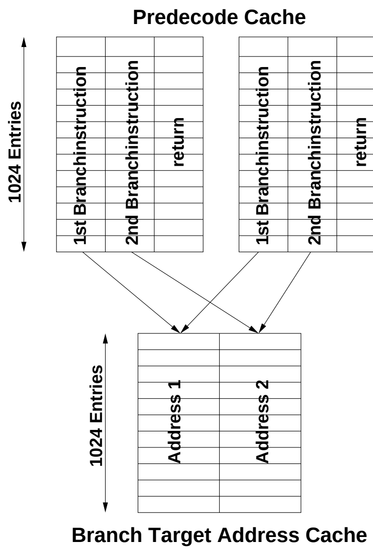


Figure A.13: Assignment of Predecode Cache Entries to Branch Target Buffer Entries [43]

Another similarity between AMD Athlon and Intel P6 is the use of a return stack buffer (cf. Section A.1.4), called *return address stack (RAS)* in the Athlon documentation. As opposed to almost all other features, it is slightly smaller than Intel's counterpart. The AMD Athlon supports only 12 `CALL`/`RET` pairs as compared to Intel's 16. Due to preemption or bad programming style, it may happen that the `CALL`/`RET` pairs in the RAS do not match. In that case, when the mismatch is detected by fetching the relevant information from the stack, a roll back like in a normal branch misprediction is initiated.

A.2.5 Built in Monitoring Support

The built in support in AMD Athlon processors, which can be utilised for the approach, is very similar to what Intel provides. The time stamp counter is also implemented in the Athlon and can be read with the same opcode as on the Intel P6. Unlike Intel's two performance monitoring counters, AMD Athlon has four of these. Since the possibilities of Athlon´s PMCs vary, the values used for the programming are also different. Additionally, the model specific registers utilised for the PMC programming have different addresses. A detailed description of the PMCs can be found in Appendix D in [4].