Uppsala Dissertations from
the Faculty of Science and Technology
36

JAKOB ENGBLOM

Processor Pipelines and Static Worst-Case Execution Time Analysis

ACTA UNIVERSITATIS UPSALIENSIS
UPPSALA 2002

Dissertation for the Degree of Doctor of Philosophy in Computer Systems presented at Uppsala University, April 19, 2002.

ABSTRACT

Engblom, J. 2002: Processor Pipelines and Static Worst-Case Execution Time Analysis. Acta Universitatis Upsaliensis. *Uppsala dissertations from the Faculty of Science and Technology* 36. 130 pp. Uppsala. ISBN 91-554-5228-0.

Worst-Case Execution Time (WCET) estimates for programs are necessary when building real-time systems. They are used to ensure timely responses from interrupts, to guarantee the throughput of cyclic tasks, as input to scheduling and schedule analysis algorithms, and in many other circumstances. Traditionally, such estimates have been obtained either by measurements or labor-intensive manual analysis, which is both time consuming and error-prone. Static worst-case execution time analysis is a family of techniques that promise to quickly provide safe execution time estimates for real-time programs, simultaneously increasing system quality and decreasing the development cost. This thesis presents several contributions to the state-of-the-art in WCET analysis.

We present an overall architecture for WCET analysis tools that provides a framework for implementing modules. Within the stable interfaces provided, modules can be independently replaced, making it easy to customize a tool for a particular target and perform performance-precision trade-offs.

We have developed concrete techniques for analyzing and representing the timing behavior of programs running on pipelined processors. The representation and analysis is more powerful than previous approaches in that pipeline timing effects across more than pairs of instructions can be handled, and in that no assumptions are made about the program structure. The analysis algorithm relies on a trace-driven processor simulator instead of a special-purpose processor model. This allows us to use existing simulators to adapt the analysis to a new target platform, reducing the retargeting effort.

We have defined a formal mathematical model of processor pipelines, which we use to investigate the properties of pipelines and WCET analysis. We prove several interesting properties of processors with in-order issue, such as the freedom from timing anomalies and the fundamental safety of WCET analysis for certain classes of pipelines. We have also constructed a number of examples that demonstrate that tight and safe WCET analysis for pipelined processors might not be as easy as once believed.

Considering the link between the analysis methods and the real world, we discuss how to build accurate software models of processor hardware, and the conditions under which accuracy is achievable.

Jakob Engblom, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden, E-mail: `jakob.engblom@docs.uu.se`*. Also at IAR Systems AB, Box 23051, SE-750 23, Uppsala, Sweden, E-mail:* `jakob.engblom@iar.se`

ISSN 1104-2516
ISBN 91-554-5228-0
Printed in Sweden by Elanders Gotab, Stockholm 2002

Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala, Sweden www.uu.se; e-mail: acta@ub.se

Acknowledgements

First of all, I would like to thank Professor *Bengt Jonsson*, my supervisor. We have both worked hard during the writing of this thesis, and without him and his keen eye for muddled thinking and hand-waving arguments, the quality of this work would have been much lower.

This thesis work was performed as an "industry PhD Student" within the Advanced Software Technology (ASTEC) competence centre (`www.astec.uu.se`) at Uppsala University funded in part by NUTEK/VINNOVA.

The *Department of Computer Systems* (DoCS, `www.docs.uu.se`) at Uppsala University represented the academic side and provided half of my financing. DoCS is now part of the Department of Information Technology (`www.it.uu.se`).

The industrial partner was *IAR Systems* (`www.iar.com`). I would like to thank IAR Systems in general, and *Olle Landström* and *Anders Berg* in particular, for giving me the opportunity to do an industry PhD. Taking on a PhD student is a big step for a small company like IAR was in 1997.

Being an industry PhD, I have traded teaching work for development work and technical training work at IAR, which I believe has been a very succesful formula. I have felt equally at home at IAR and at the university, and hopefully I have helped increase the flow of information and ideas between industry and academia. For me personally, this duality has been very inspiring and rewarding. Despite the principle of no teaching, I have taken the chance to get a little involved in undergraduate teaching anyway, giving guest lectures for a number of real-time and computer architecture courses.

Andreas "Ebbe" Ermedahl has been my team-mate in WCET analysis research since I started working in the project in 1997. Together, we have achieved much more than any one of us could have done by himself. I thank Ebbe for years of intense and inspiring cooperation and discussion.

Friedhelm Stappert at C-Lab in Paderborn, Germany, joined our project in 1999, adding fresh perspectives and implementation manpower. Despite the

ASTEC Advanced Software Technology

ARTES A network for Real-Time research and graduate Education in Sweden

IAR SYSTEMS

geographical distribution, Ebbe, Friedhelm, and I have managed to produce a prototype tool and a number of papers together.

Professor *Hans Hansson* got me into real-time systems and WCET research via my Master's thesis project back in 1997, and has been a great help in writing papers and discussing ideas ever since.

Jan Gustafsson has provided valuable and constructive feedback on this thesis and earlier papers, and has been a valuable partner in the research for a long time.

Carl von Platen at IAR has been team leader for me and the other industry PhD students at IAR (Jan Sjödin and Johan Runesson) for the past few years, and I thank him for many invigorating discussions.

Some of our research in WCET analysis and its applications has been performed with the help of hard-working Master's Thesis students: *Sven Montàn*, *Martin Carlsson*, and *Magnus Nilsson*. Thanks to them, several interesting ideas have been explored that we would not have had time for otherwise. *Jan Lindblad* at Enea OSE Systems (`www.ose.com`) and *Jörgen Hansson* at CC-Systems (`www.cc-systems.se`) have been kind enough to help finance some of the students and have provided industrial input to our research.

Over the years, I have had many interesting discussions with people in industry and academia (not all on WCET analysis). I cannot list them all, but some of them are Professor *David Whalley* at Florida State University, *Iain Bate* in York, Professor *Sang Lyul Min* at Seoul National University, *Stefan Petters* in München, *Peter Altenbernd* at C-Lab in Paderborn, *Thomas Lundqvist* at Chalmers, Professor *Erik Hagersten* here in Uppsala, and *Raimund Kirner* and *Pavel Atanassov* at TU Wien.

Björn Victor helped me set up the LaTeXdocuments, and *Johan Bengtsson* and I helped each other figure out how to publish our theses and how and where to print them.

I would also like to thank everyone at *IAR* in Uppsala and *DoCS* for all the fun we have had together!

Anna-Maria Lundins Stipendiefond at *Smålands Nation* has provided generous grants of travel money. If you are planning on getting a PhD in Uppsala, Smålands is the best nation!

The *ARTES Network* (`www.artes.uu.se`) has provided travel funding and some nice summer schools.

Of course, life in Uppsala would have been much less fun without all of my wonderful *friends*.

I would also like thank my *parents*, Lars-Åke and Christina, and my brother Samuel and sister Cecilia. They have always been there for me!

And a special thanks goes to that very special person who has brightened my life in the past few years and who supported me in the stressful spurt of my thesis writing, *Eva*.

Contents

Publications by the Author

Since I started working within the WCET project in 1997, I have published several papers in cooperation with various people. The following is a list of all publications that have been subject to peer review:

A. Jakob Engblom, Andreas Ermedahl, and Peter Altenbernd: Facilitating Worst-Case Execution Times Analysis for Optimized Code. In *Proceedings of the 10th Euromicro Real-Time Systems Workshop*, Berlin, Germany, June 1998.

B. Jakob Engblom: Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*, Atlanta, Georgia, USA, May 1999.

C. Jakob Engblom: Static Properties of Commercial Embedded Real-Time Programs, and Their Implication for Worst-Case Execution Time Analysis. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, Vancouver, Canada, June 1999.

D. Jakob Engblom and Andreas Ermedahl: Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proceedings of the 6th Internation Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, Hong Kong, China, December 1999.

E. Jakob Engblom and Andreas Ermedahl: Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, Orlando, Florida, USA, December 2000.

F. Jakob Engblom: Getting the Least Out of Your C Compiler. Class and paper presented at the *Embedded Systems Conference San Francisco (ESC SF)*, April 2001.

G. Jakob Engblom, Andreas Ermedahl and Friedhelm Stappert : A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems. Presented at the *Workshop on Real-Time Tools (RT-TOOLS 2001)*, held in conjunction with CONCUR 2001, Aalborg, Denmark, August 2001.

H. Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom : Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the 4th International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2001)*, Atlanta, Georgia, USA, November 2001.

I. Jakob Engblom: On Hardware and Hardware Models for Embedded Real-Time Systems. Short paper presented at the *IEEE Embedded Real-Time Systems Workshop* held in conjunction with the *22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, London, UK, December 2001.

J. Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson: Execution-Time Analysis for Embedded Real-Time Systems. Accepted for publication in the *Software Tools for Technology Transfer (STTT)* special issue on ASTEC (forthcoming).

Paper D presents an early version of the pipeline analysis and modeling methods presented in this thesis (Chapters 4, 6, and 7).

Papers G and J present the WCET tool architecture in Chapter 3.

Papers B and C present a prestudy concerning the structure of embedded software, as referenced in Chapter 1.

Paper E and H deals with how to represent program flows and how to take advantage of the knowledge for WCET analysis, as discussed in Section 2.2 and Section 2.4.

Paper A is a summary of my Master's thesis [Eng97] dealing with the mapping problem discussed briefly in Section 2.2.4.

Paper I is a compressed version of the discussion on how to build quality processor simulators given in Chapter 8.

Paper F is a practical guide in how to write efficient C code for embedded systems, and has been presented in various versions at a number of industry conferences.

In addition to the above papers, I have been involved in the creation of a number of technical reports [Eng98, EES+99, SEE01a, EES01b] and work-in-progress papers [EES00, ESE00].

Compared to our previous publications, there is quite a lot of new material in this thesis. In particular, the discussion about properties of pipelines in Chapter 5 is completely new. The extension conditions given for the timing analysis in Chapter 6 are different from those presented in Paper D. Only a small part of the material in Chapter 8 was presented in Paper I. Most of the experimental results presented in Chapter 9 are also new.

Almost all of my research has been carried out within the framework of the ASTEC WCET project in close collaboration with several colleagues. The prototype implementation and experimentation has been carried out in cooperation with Andreas Ermedahl (also at Uppsala University) and Friedhelm Stappert (at C-Lab in Paderborn, Germany).

Chapter 1

Introduction

This thesis is about worst-case execution time (WCET) analysis for embedded systems, in particular about the effects of processor pipelines on the WCET. Before I present the concrete contributions of this thesis, I think it is appropriate to provide some background on real-time systems, embedded systems, processor pipelines, and other material of relevance. If you think you know the background already, feel free to skip ahead to Section 1.11 where the contributions are presented.

1.1 Real-Time Systems

A real-time system is a computer-based system where the *timing* of a computed result is as important as the actual *value*. Timing behavior is much less understood than functional behavior, and one of the most common causes of unexpected failures.

Real-time does not mean that a value should be produced as quickly as possible: in most cases, steady and predictable behavior is the desired property. Consider video and audio playback: the important consideration is the steady generation of images and synchronized sound, at a pace consistent with the recording speed of the video and audio. Being too slow is obviously bad, but being too fast, i.e. playing video faster, is not good either. The key is to be *just right*.

A distinction is usually made between *soft* and *hard* real-time systems. In hard real-time systems, the failure to meet a *deadline* (the time limit allocated to complete a computation) can be fatal, like braking a car too late or letting a chemical process run out of control.

In soft real-time systems, on the other hand, an occasional failure to meet a deadline does not have permanent negative effects. Video playback is a good example: skipping the occasional frame is not fatal, and often not even detectable by the user. In general, for soft real-time systems, the failure to meet deadlines means that the quality of the service provided is reduced, but the system still provides a useful service.

To guarantee the behavior of a hard real-time system, the *worst case* behavior of the system has to be analyzed and accounted for. If a system has several concurrent programs running, it has to be shown that all programs can meet their respective deadlines even in the case that all programs simultaneously perform the greatest amount of work. In this thesis, we are mostly dealing with hard real-time systems, even if the techniques presented can be useful for the development of soft real-time systems as well.

A good example of a hard real-time system are the devices that protect transformers from damage caused by lightning strikes in powerlines. Such a system has to detect a lightning strike within a millisecond and take the transformer offline, or the transformer will catch fire. If it meets its deadline, it succeeds in its task. If the deadline is not met, something very expensive gets blown up. Note that no extra value is obtained from being faster then required to meet the deadline.

Some hard real-time systems has the additional requirement that the *variance* (jitter) in the computation should be as small as possible. For example, in control systems like engine controllers, the results of the computation of control algorithms should be generated after a fixed time has passed from the measurements used in the computation, as this is necessary to maintain good controller performance.

1.2 Embedded Systems

An embedded system is a computer that “does not look like a computer”. Instead, it is embedded as a component in a product. It is a computer used as a means to achieve some specific goal, not a goal in itself. Today, embedded systems are everywhere: about eight billion embedded processors are sold each year, and they are finding their way into more and more everyday items. In recent years, 98%-99% of the total number of processors produced have been used in embedded systems [Hal00b, Had02][1].

For example, a modern car like the Volvo S80 contains more than thirty embedded processors, communicating across several networks [CRTM98, LH02]. A GSM mobile phone contains at least two processors: a digital signal processor (DSP) to handle encoding and decoding of speech and data signals and a main processor to run the menu system, games, and other user-interface functions. Household items like microwave ovens contain simple processors. Embedded computers control chemical processes and robots in manufacturing plants. Modern instable jet fighters like the SAAB JAS 39 Gripen are completely dependent on their embedded control systems in order not to crash.

[1]However, desktop processors represent a much larger share of the revenues in the processor market, since the per-chip costs is on the order of dollars in the embedded field but on the order of hundreds of dollars in the desktop field.

Most embedded systems are (hard) real-time systems, since they are part of devices interacting with and controlling phenomena in the surrounding physical world. There are non-embedded real-time systems like multimedia players for PCs (PC-based real-time systems are mostly soft real-time systems), and there are non-real-time embedded systems like toys, but the embedded hard real-time systems are far more common. Accordingly, we have focussed on the needs of the developers of embedded hard real-time systems.

1.3 Execution Time and Real-Time Systems

The timing of a real-time system has to be validated on a system level: only if each and every component of the system fulfill their timing requirements can we be sure that the complete system meets its requirements.

For a system involving software programs (as all embedded computer systems do), we need to determine the timing behavior of the programs running on the systems. The timing of the programs are then used to determine the behavior of the complete system (see Section 1.5). Knowing the execution time properties of your code is one of the most important parts of real-time systems development, and failing to ascertain the timing is a quick way to system failure [Gan01, Ste01].

A software program typically does not have a single fixed execution time, which is unfortunate for the predictability of a system. *Variation* in the execution time occurs because a program might perform different amounts of work each time it is executed, or because the hardware it executes on varies in the amount of time required to perform the same set of instructions. This variability in the execution time of programs has to be analyzed in order to construct reliable embedded real-time systems.

Note that a program with high execution time variation can still be considered *predictable*, if we can model and predict the causes of the variation in execution time. Typically the control flow of a program can be modeled with reasonable precision, while the hardware can pose a very big problem and give rise to actual unpredictability (the control flow aspects are discussed in Section 1.9.2 and Section 2.2.2, and the hardware aspects in Chapter 8 and Chapter 10).

The implication is that real-time systems involving embedded computers have to be analyzed as a *combination* of software and hardware. Both the properties of the hardware and of the software have to be accounted for in order to understand and predict the behavior of the programs running on the system as well as the complete system. The use of intermediate software like real-time operating systems can facilitate such analysis, but in the end, the actual software and hardware being part of a shipping product have to be analyzed as a whole.

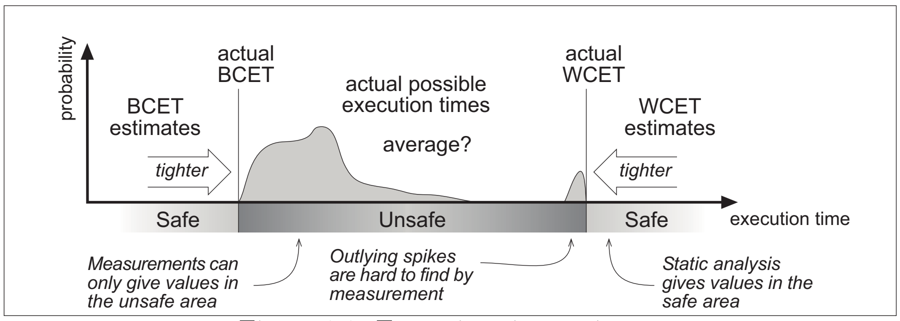

probability
actual BCET
actual possible execution times
average?
actual WCET
BCET estimates
WCET estimates
tighter
tighter
Safe
Unsafe
Safe
execution time
Measurements can only give values in the unsafe area
Outlying spikes are hard to find by measurement
Static analysis gives values in the safe area

Figure 1.1: Execution time estimates

1.4 Execution Time Estimates

There are a number of different execution time measures that can be used to describe the timing behavior of a program. The *worst-case execution time*, WCET, is the longest execution time of the program that will ever be observed when the program runs in its production environment. The *best-case execution time*, BCET, is the shortest time the program will ever take to execute. The *average execution time* is the average, which lies somewhere between the WCET and the BCET. It is in general very hard to determine the exact actual WCET (or BCET) of a program, as this depends upon inputs received at run time, and the average is even more difficult to determine since it depends on the distribution of the input data and not just the extremes of program behavior.

Figure 1.1 shows how the BCET and WCET relate to the execution time of a program. The curve shows the probability distribution of the execution time of a program. There is an upper bound beyond which the probability of the execution time is zero, the actual WCET, and lower bound, the actual BCET.

Timing analysis aims to produce *estimates* of the WCET and BCET. A timing estimate must be *safe*, which means that WCET estimates must be greater than, or, in the ideal case, equal to the actual WCET (the righthand area marked "safe"). Conversely, the BCET estimate has to be less than or equal to the actual BCET (lefthand "safe" area). Any other WCET or BCET estimate is unsafe. An underestimated WCET is worse than no WCET estimate at all, since we will produce a system which rests on a false assumption, and which we believe is correct, but that *can* fail.

Note that it is trivial to produce conservative but perfectly useless estimates. A statement like "the program will terminate within the next 5 billion years" is certainly true (unless the program contains an infinite loop), but not very useful (dimensioning for a WCET like this would be a tremendous waste of resources). To be useful, the estimate must not only be conservative, but also *tight*, i.e. , close to the actual value, as shown by the "tighter" arrows in Figure 1.1.

1.5 Uses of WCET

The concept of the worst-case execution time for a program has been around in the real-time community for a long time, especially for doing schedulability analysis and scheduling [LL73, ABD$^+$95, CRTM98]. Many scheduling algorithms and all schedulability analysis assume knowledge about the worst-case execution time of a task. However, WCET estimates have a much broader application domain; whenever timeliness is important, WCET analysis is a natural technique to apply.

For instance, designing and verifying systems where the timing of certain pieces of code is crucial can be simplified by using WCET analysis instead of extensive and expensive testing. WCET estimates can be used to verify that the response time of an interrupt handler is short enough, that a system reacts quickly enough, or that the sample rate of a control loop or encoder/decoder is kept under all circumstances.

Tools for modeling and verifying systems modeled as timed automata, like Uppaal [LPY97], HyTech [HHWT97], and Kronos [BDM$^+$98] can use WCET estimates to obtain timing values from the real implementation of a system [BPPS00].

When developing embedded systems using graphical programming tools like IAR visualSTATE, Telelogic Tau, and I-Logix StateMate, it is very helpful to get feedback on the timing for model actions and the worst-case time from input event to output event, as demonstrated by Erpenbach et al. [ESS99]. Kirner et al. perform WCET analysis for C code generated from Matlab/Simulink models, and back-annotate the results into the model [KLP01].

WCET analysis can be used to assist in selecting appropriate hardware for a real-time embedded system. The designers of a system can take the application code they will use and perform WCET analysis for a range of target systems, selecting the cheapest (slowest) chip that meets the performance requirements, or adjust the clock frequency based on the worst-case timing estimate.

The HRT real-time system design methodology defined by British Aerospace Space Systems makes use of WCET values to form execution time skeletons for programs, where WCET estimates are given for the code that executes between accesses to shared objects [HLS00a].

WCET estimates on the basic-block level can be used to enable host-compiled time-accurate simulation of embedded systems. Programs are compiled to run on a PC, with annotations inserted on each basic block to count time as it would evolve on the target system, allowing the PC to simulate the timing of the target system [Nil01].

Timing estimates on the basic-block level can also be used to interleave the code of a background task with the code of a foreground program, maintaining the timing of the background task without the overheads of an operating system to switch between the tasks [DS99].

In most real-time systems, best-case execution time (BCET) estimates are also very interesting, since the variation in execution time between best and worst cases is often what causes fatal timing errors in a system (when the program suddenly runs much faster or slower than expected from previous observations) [Gan01]. In many cases, code with constant timing is what is sought, simply to make the system more predictable [Eyr01]. For tasks with a high variation in execution time between best, average, and worst, using the WCET for scheduling will give low resource utilization [Mär01].

For soft real-time applications, average execution time estimates are important, since they help estimate the achievable sustained throughput (number of phone calls a switch can handle, the frames per second achievable in a computer game, etc.). Occasional spikes in execution time are not as critical. But even for soft real-time systems, WCET analysis can still be used to indicate potential bottlenecks in programs, even though the WCET estimate as such is not of much use for system-level analysis. Also, since a missed deadline does correspond to reduced quality of service, WCET estimates can still be useful to maximize the quality of service for a specified load.

1.6 Obtaining WCET Estimates

There are two ways to obtain worst-case execution time information for software: *measure* it experimentally, or estimate it by *static analysis*.

The state of the practice in WCET estimation today is *measuring* the run time of a program, running it with "really bad" input (or a set of "typical" inputs), and keeping track of the worst execution time encountered ("high-water marking"). Then, some safety margin is added, and hopefully the real worst case lies inside the resulting estimate. However, there are no guarantees that the worst case has indeed been found, and measurements can only produce statistical evidence of the probable WCET and BCET, but never complete certainty. Note that the case illustrated in Figure 1.1, where the WCET is an outlier value, is quite common in practice, which makes measuring the WCET riskier [Gan01].

To obtain a safe WCET estimate, we must use mathematically founded *static analysis* methods to find and analyze all possible program behaviors. In static analysis, we do not run the program and measure the resulting execution time; instead, the code of the program (source code and executable code) is inspected and its properties determined, and a worst-case execution time estimate is generated from the information. Static analysis allows us to overcome the measurement difficulties posed by execution time variability, provided that we manage to model the relevant input data variation and hardware effects, and use effective analysis methods.

The WCET of a program depends both on the control flow (like loop iterations, decision statements, and function calls), and on the characteristics of the target hardware architecture (like pipelines and caches). Thus, both the control flow and the hardware the program runs on must be considered in static WCET analysis. WCET approaches that work by ascribing a certain execution time to each type of source code statement will not work when using anything but a

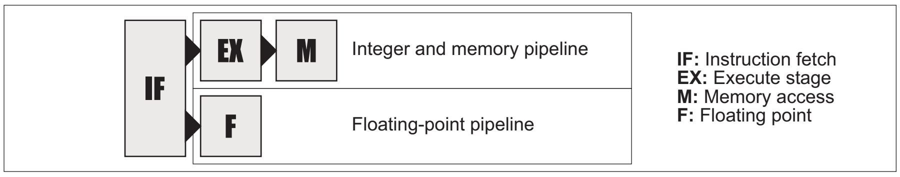


Figure 1.2: Example scalar pipeline with parallel units

trivial compiler and simple hardware.

It should be noted that static WCET analysis carries the distinct advantage over measurement techniques that more complex CPU architectures can be analyzed safely. The greater hardware variability introduced by such architectures is hard to account for in measurement, while being modeled by necessity in static WCET analysis. Another advantage is that the target hardware does not have to be available in order to obtain timing estimates, since static WCET analysis typically uses a hardware model to perform the analysis and not the actual hardware.

In principle, static WCET analysis can be carried out by hand, without any tool support. However, this is only practical for small and simple programs, executed on simple hardware. Thus, *automated tools* are crucial to make it practical to apply static WCET analysis. Widespread use of static WCET analysis tools would offer improvements in product quality and safety for embedded and real-time systems, and reduce development time since the verification of timing behaviour is facilitated. This thesis presents some steps towards such a tool, especially considering the timing effects of processor pipelines.

1.7 Processor Pipelines

The largest part of this thesis is devoted to the problems of analyzing the timing behavior of *processor pipelines*. The purpose of employing a pipeline in a processor is to increase performance by overlapping the execution of successive instructions. Early computers executed one instruction at a time, reading it from memory, decoding it, reading the operands from memory or registers, carrying out the work prescribed, and finally writing the results back to registers or memory. In the late 1950's, it became clear that these phases could be overlapped, so that while one instruction was being decoded, the next instruction could be fetched, etc., and thus the concept of *pipelined* execution was born. The concept is similar to that of a assembly line where a car is assembled piece by piece at several different assembly stations. The first commercial computer to use a pipeline is considered to be the IBM 7030 "Stretch", launched in 1959. It took until the 1980's and the first wave of RISC processors for pipelines to be used in microprocessors and personal computers, and until the 1990's until they were used in embedded systems. Today, pipelines are almost mandatory in new CPU designs [HP96].

A pipeline consists of a number of *stages* that an instruction goes through in order to execute. Not all instructions have to go through all stages, and it is quite common for pipelines to have several parallel paths that instructions can take. Figure 1.2 shows a processor containing four stages. In the `IF` stage, instructions are fetched from memory. Integer and data memory instructions then take the path through the `EX` stage, where arithmetic operations are performed, and the `M` stage, where data memory is accessed. Floating-point instructions execute in the `F` stage, in parallel to the integer and data memory instructions. Only one instruction can use a pipeline stage at any one time, and execution progresses by instructions moving forward in the pipeline, entering and leaving successive stages until they finally leave the pipeline.

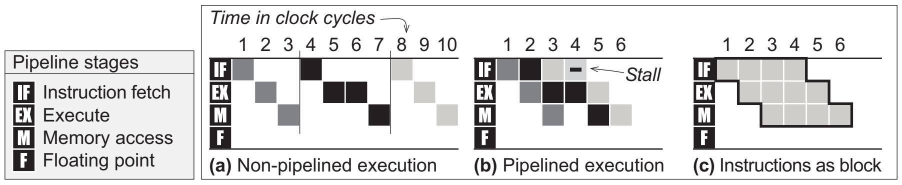


Figure 1.3: Pipelining of instruction execution

Figure 1.3 shows the overlap between instructions achieved in a pipeline, using *pipeline diagrams* similar to the *reservation tables* commonly used to describe the behavior of pipelined processors [Dav71, Kog81, HP96]. Time runs on the horizontal, with each tick of time corresponding to a processor clock cycle. The pipeline stages are shown on the vertical. Instructions progress from upper left to lower right, and each step of execution is shown as a square.

Figure 1.3(a) shows how three instructions execute in a processor without pipelining, each finishing its entire execution before the next instruction can start, with a total execution time of ten cycles. Note that an instruction can spend more than one cycle in a certain pipeline stage. Figure 1.3(b) shows the same instructions overlapped by the pipeline, generating a total execution time of only six cycles. Usually, we deal with blocks of instructions, and Figure 1.3(c) shows how the instructions in the example are grouped into a block, where we do not distinguish the individual instructions anymore.

Figure 1.4 shows the style of most illustrations in this thesis: only blocks of instructions are shown, not the constituent instructions. Each block is given

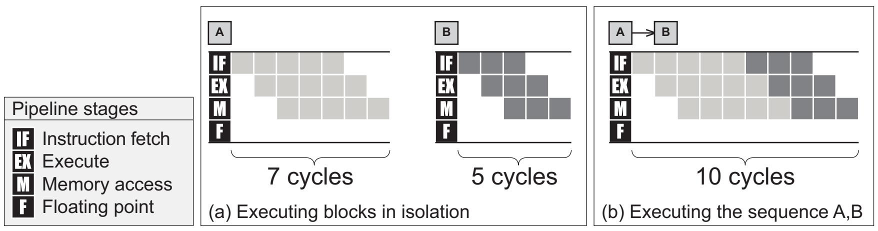


Figure 1.4: Pipelining basic blocks

its own color to distinguish them from each other. Since a block can contain a single instruction, we often use "instruction" and "block" interchangeably.

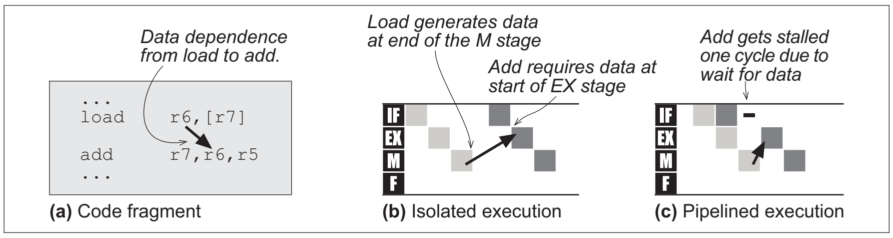


Figure 1.5: Data hazard causing pipeline stall

Notice how the third instruction in Figure 1.3 has to wait before entering the `EX` stage because the second instruction needs the `EX` stage for two cycles, causing a *pipeline stall* where the instruction waits in the `IF` stage without doing any work. Pipeline stalls like this, where an instruction cannot enter its next stage because another instruction is using that stage, are called *structural hazards*. Note that if an instruction is stalled in the `IF` stage, no following instruction can enter the pipeline until the stall has cleared. Stalls can also appear because an instruction requires some data generated by a previous instruction, and when, because of the pipelining of execution, the required information is not yet available when the instruction needs it. This is called a *data hazard*, and an example is given in Figure 1.5.

Type	**# pipelines**	**issue order**	**instr/cycle**	**scheduling**
Simple scalar	1	in-order	1	static
Scalar	>1	in-order	1	static
Superscalar in-order	>1	in-order	>1	dynamic
VLIW	>1	in-order	>1	static
Superscalar out-of-order	>1	out-of-order	>1	dynamic

Figure 1.6: A simple classification of pipelines

Pipelines come in a wide range of complexities. In general, to reach higher performance, more complex pipelines are required. Figure 1.6 shows the classification of pipeline types employed in this thesis, and their defining properties. In-order issue means that instructions are sent to the pipeline in the order specified in the program, while out-of-order means that the processor can potentially change the order of the instructions to make the program execute faster.

1.7.1 Simple Scalar Pipelines

The simplest form of a pipeline is a single pipeline where all instructions execute in order, as employed on the early SPARC and MIPS processors. An example of such a pipeline is shown in Figure 1.7. The stages in this pipeline are `IF`, where instructions are fetched, `ID`, for decoding the instructions, `RR`, where operands are read from registers, `EX`, where the results of arithmetic instructions

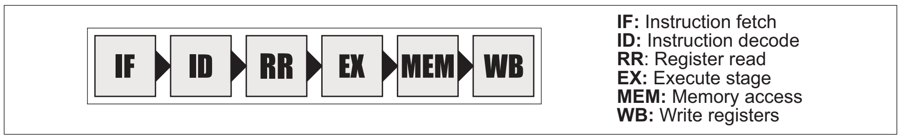


Figure 1.7: Example simple scalar pipeline

are generated, `MEM`, where memory is accessed for data, and `WB`, where values computed in `EX` or read from memory in `MEM` are written back to registers. All instructions must progress through all the stages in this pipeline in order to complete execution.

Current industrial examples of simple scalar pipelines are the ARM7 and ARM9 cores from ARM [ARM95, ARM00b], the Hitachi SH7700 [Hit95], Infineon C167 [Inf01], ARC cores [ARC], and some embedded MIPS cores [Sny01]. The IBM 7030 also belonged to this family, using a four-stage pipeline [HP96]. Such simple pipelines can reach respectable performance while still being simple enough to implement in a very small and cheap processor. The number of pipeline stages vary between three and about ten, with most processors having between five and seven stages.

1.7.2 Scalar Pipelines

To increase the performance of a processor, execution can be split across multiple pipelines at some stage in the execution, as illustrated in Figure 1.2. Prior to the split points, instructions proceed through a common sequence of stages. For example, beginning in the late 1980's, many processors added a second pipeline to support floating point instruction execution in parallel with integer instructions [Pat01]. A second pipeline can also be used for other purposes than floating point, as illustrated by the NEC V850E [NEC99] (see Figure 7.1).

Industrial examples of this type of pipeline are the NEC V850E [NEC99], MIPS R3000 [LBJ$^+$95], and MIPS R4000 [Hei94].

1.7.3 Superscalar In-Order Pipelines

A *superscalar CPU* allows several instructions to start each clock cycle, to make more efficient use of multiple execution pipelines. Instructions are grouped for execution *dynamically*: it is not possible to determine which instructions will be issued as a group just by inspecting the program text, since the behavior of the instruction scheduler must be taken into account. Note that instructions are still issued in-order, only in groups of (hopefully) several instructions per cycle.

Figure 1.8 shows a simplified view of the pipeline of a superscalar processor. The `IF` stage fetches several instructions per clock cycle from memory, and the `S` stage maintains a queue of instructions that are waiting to issue, finding groups of instructions that can be executed in parallel. Each of the functional

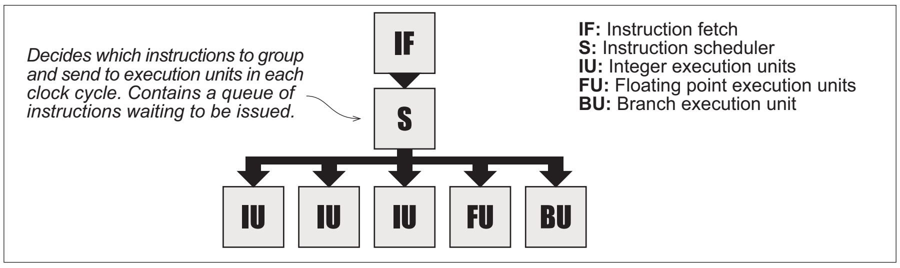

Decides which instructions to group and send to execution units in each clock cycle. Contains a queue of instructions waiting to be issued.
IF
S
IU
IU
IU
FU
BU
IF: Instruction fetch
S: Instruction scheduler
IU: Integer execution units
FU: Floating point execution units
BU: Branch execution unit

Figure 1.8: Example in-order superscalar pipeline structure

units `IU`, `BU`, and `FU` are usually multi-stage pipelines in their own right. On modern processors, total pipeline depth from instruction fetch to completion of execution can be twenty cycles or more.

This type of pipeline has been used in many early (and current) 64-bit server processors like the HP-PA 7200 [CHK$^+$96], Alpha 21164 [DEC98], SuperSparc I [SF99], and UltraSparc 3 [Son97].

1.7.4 VLIW (Very Long Instruction Word)

In a Very Long Instruction Word (VLIW) processor, instructions are statically grouped for execution at compile time. The processor fetches and executes very long instruction words containing several operations that the compiler has determined can be issued independently [SFK97, pp. 93–95]. Each operation bundle will execute all its operations in parallel, without any interference from a hardware scheduler. The structure is similar to that of an in-order superscalar, but without the complexity of the dynamic instruction grouping.

Current examples of this class of processors are mainly DSPs, like the Texas Instruments TMS320C6xxx, Analog Devices TigerSharc, and Motorola/Lucent StarCore SC140 [WB98, Hal99, TI00, Eyr01]. On the desktop and server side, the Intel/HP Itanium is a VLIW processor [Int00b], and the core of the Transmeta Crusoe is also a VLIW design [Kla00].

1.7.5 Superscalar Out-of-Order Pipelines

To get the most performance out of multiple pipelines, modern high-end superscalar processors allow instructions to execute *out-of-order* (in an order different from that of the program text). This allows the processor to use the pipelines more efficiently, mainly since delays due to data dependences can be hidden by executing other instructions while the pipeline waits for the instruction providing the information to finish.

However, out-of-order execution introduces a great deal of complexity into the pipeline for tracking the instructions that are executing, and determining which instructions can be executed in parallel. Such processors are optimized for providing good average-case performance, but the worst-case behavior can be very hard to determine and analyze [BE00, Eyr01].

Examples of processors employing out-of-order issue are the Pentium III and Pentium 4 processors from Intel [Int01a], AMD's Athlon [AMD00], the PowerPC G3 and G4 from Motorola and IBM [Mot97], HP's PA-8000 [Kum97], and the Alpha 21264 from Compaq [Com99]. The first commercial processor incorporating out-of-order dynamic scheduling of instructions was the CDC 6600, launched in 1964. The technology entered high-end microprocessors in the mid-1990's [Pat01].

Note that out-of-order execution is not absolutely necessary for high performance, as proven by the UltraSparc 3 (in January 2002, the 1050 Mhz UltraSparc 3 (Cu) processor had the third highest SPECfp2000 peak score, after IBM's 1300 Mhz Power4 and Compaq's 1000 Mhz Alpha 21264. It was ahead of the aggressive out-of-order Intel Pentium 4 processor (running at 2000 Mhz) and AMD's Athlon XP (at 1600 Mhz)).

1.8 Properties of Embedded Hardware

An embedded system is usually based on a *microcontroller*, a microprocessor with a set of peripherals and memory integrated on the processor chip. Microcontrollers can be standard off-the-shelf products like the Atmel AT90 line or Microchip PIC, or custom application-specific integrated circuits (ASIC) based on a standard CPU core (like an ARM or MIPS core) with a custom set of peripherals and memory on-chip.

As shown in Figure 1.9, microcontrollers completely outnumber desktop processors in terms of units shipped. We also note that simple microcontrollers (8-bit and 16-bit) dominate sales. The reason for this is that embedded systems designers, in order to minimize the power consumption, size, and cost of the overall system, use processors that are just fast and big enough to solve a problem.

Processor Category	**Number Sold**
Embedded 4-bit	2000 million
Embedded 8-bit	4700 million
Embedded 16-bit	700 million
Embedded 32-bit	400 million
DSP	600 million
Desktop 32/64-bit	150 million

Figure 1.9: 1999 world market for microprocessors [Ten99]

For most 4-, 8-, and 16-bit processors, low-level WCET analysis is a simple matter of counting the executing cycles for each instruction, since these CPUs are usually not pipelined. There are some 16-bit processors with pipelines (like the Infineon C166/C167).

Processor Family	**Number Sold**
ARM	151 million
Motorola 68k	94 million
MIPS	57 million
Hitachi SuperH	33 million
x86	29 million
PowerPC	10 million
Intel i960	7.5 million
SPARC	2.5 million
AMD 29k	2 million
Motorola M-Core	1.1 million

Figure 1.10: 1999 32-bit microcontroller sales [Hal00b]

32-bit processors are usually more complex, but the embedded 32-bit processors still tend to be relatively simple. Figure 1.10 shows market shares for 1999 in the 32-bit embedded processor segment. It is clear that simple architectures dominate the field. The best-selling 32-bit microcontroller family is the ARM from Advanced Risc Machines. All ARM variants have a single, simple pipeline, and very few have caches. The embedded x86 and PowerPC processors belong to desktop processor families, but the processors used for embedded systems are usually based on older 386 and 486 designs (for the x86), or simplified systems with a scalar pipeline (for the PowerPC). Cutting-edge processors are simply too expensive and power-hungry to be used in most embedded systems.

In 2000 and 2001, ARM has dominated the embedded 32-bit RISC market. In 2001, ARM sold about 400 million units[2], with MIPS following at 60 million units, and Hitachi/ST Microelectronics SH at 49 million units. PowerPC brings up the rear with 18 million units, and all other architectures share the remaining 11 million units [Ced02]. Note that these statistics cover a smaller part of the overall market than those in Figure 1.10, since not all processors qualify as "RISC" (the x86 and Motorola 68k are not part of the embedded RISC market).

Digital Signal Processors (DSPs), processors built for maximal performance on signal processing and media processing tasks, are rapidly expanding their market due to the needs of portable media devices. DSPs are becoming more complex to meet higher computational demands, but are still built for predictable performance. They usually employ very long instruction word (VLIW) architectures to boost performance rather than going for dynamic out-of-order scheduling in hardware [Eyr01].

For embedded systems designed for predictability, memory is generally based on on-chip static RAMs, since caches are considered to introduce too much variability in the system. Note that even for systems that have not been explicitly designed for predictability, caches are still not that common. Caches are used

[2]A very impressive increase in total sales compared to 1999, mainly thanks to ARM becoming the dominant architecture for PDAs and high-end mobile phones.

on many high-end embedded 32-bit CPUs, but are generally not needed for processors running at clock frequencies below 100 MHz. Also, caches are quite demanding in terms of processor area and power consumption, making them hard to squeeze into the limited budgets of embedded processors.

Accordingly, the embedded real-time systems market requires WCET analysis methods that are easy to port to several architectures and that support the efficient handling of on-chip memory and peripherals, and the inclusion of cache analysis results. One should note that in order to build a reliable real-time system, the selection of hardware is critical. This issue is discussed at some length in Chapter 8 and Chapter 10.

1.9 Properties of Embedded Software

A useful WCET analysis tool has to be adapted to the characteristics of programs used in embedded real-time systems. Here, we will discuss two categories of properties: which types of constructions are used when writing program code, and which types of algorithmic behavior can be expected?

1.9.1 Programming Style

Today, most embedded systems are programmed in C, C++, and assembly language [SKO$^+$96, Gan02]. More sophisticated languages, like Ada and Java, have found some use, but the need for speed, portability (there are C compilers for more architectures than any other programming language), small code size[3], and efficient access to hardware is likely to keep C the dominant language for the foreseeable future. C is also very popular as a backend language for code generation from graphical programming notations such as UML, SDL, and StateCharts, which are getting increasingly important because they promise higher programming productivity and higher-quality software.

As a prelude to this research, we performed an investigation into the properties of embedded software. The programs investigated were written in C and used in actual commercial systems [Eng98, Eng99a]. In general, embedded programs code seems to be rather different from desktop code. Desktop software has a tendency to perform arithmetic operations, while embedded software contains more logical and bitwise operations. This means that we should not use desktop code like the SpecInt benchmarks to test tools intended for embedded systems, but rather look for benchmarks relevant to the particular application area [Eng99b, Hal00a].

Concerning static WCET analysis, the result of our investigation was that while most of the code is quite simple (using singly nested loops, simple deci-

[3]It is a common claim that Java programs could be quite small thanks to the byte-code architecture, but if the memory required for the Java Virtual Machine is factored in the total system size is still quite large. Also, Java compilers do not have the same level of size optimization as good C compilers for embedded targets.

sion structures, etc.), there are some instances of highly complex control flow. For instance, deeply-nested loops and decision structures do occur, and, more problematically, recursion and unstructured code. From a low-level timing analysis perspective, this means that we cannot assume any particular structure for program (like perfectly nested loops) if we want to stay general [Eng99a].

Usually, WCET analysis is applied to user code, but in any system where an operating system is used, the timing of operating system calls have to be taken into account. This means that WCET analysis must also consider operating system code. Colin and Puaut [CP01b] have investigated parts of the code for the RTEMS operating system, and found no nested loops, unstructured code, or recursion. This seems to be a well-behaved special case, since Carlsson [Car02] reports rather more complex program structures in the OSE Delta kernel.

Holsti et al. [HLS00a] report that a compiler used for compiling real-time software for a space application employed several assembly-written libraries that included features like jumps to function pointers and unstructured loops; this has to be addressed in a practical WCET tool.

Another important aspect of software structure is that ordinarily, only small parts of the applications are really time-critical. For example, in a GSM mobile phone, the time-critical GSM coding and decoding code is very small compared to the non-critical user interface. Using this fact, ambitious WCET analysis can be performed on the critical parts (provided that they can be efficiently identified and isolated from the rest of the code), ignoring the rest of the code.

1.9.2 Algorithmic Timing Behavior

Regarding the timing behavior of the algorithms used in embedded real-time programs, the general opinion in the real-time systems field is that programs should be written in such a way that termination is guaranteed and variability in execution time is minimized.

Ernst and Ye [EY97] note an interesting property of the control flow of some signal-processing algorithms. While the program source code contains lots of decisions and loops, the decisions are written in such a way that there is only a single path through the program, regardless of the input data[4]. Similarly, Patterson [Pat95] notes that most branches in the Spec benchmarks are independent of the input data of the program.

Indicating the opposite, Fisher and Freudenberger [FF92] note a large variability in the number of instructions between "unpredictable" branches, in an experiment designed to test the effectiveness of profile-driven optimization, indicating that the predictability varies quite significantly between programs.

Ganssle [Gan01] reports that in many instances on small 8-bit and 16-bit microcontrollers, compiler libraries for arithmetic functions like multiplication exhibit large execution time variation, and recommend that algorithms with

[4]A typical example of this is when decision statements depend on the value of a loop counter, like performing different work for "odd" and "even" iterations of a loop.

more stable timing is used. For WCET analysis, such variability will contribute to making the estimates quite pessimistic visavi the common case, and the use of stable algorithms seems quite valuable. The same behavior is observed in hash table algorithms by Friedman et al. [FLBC01].

Hughes et al. [HKA$^+$01] have investigated the execution time variability inherent in modern multimedia algorithms. They found that the algorithms had a large execution time variability across different types of frames, with smaller variation within each type of frame.

Taken together, the research cited indicates that there are programs where the flow is very predictable, which a WCET analysis method should take advantage of, but that cases will also occur where not much can be known about the program flow, and thus it is dangerous to base a tool design on programs having very predictable flow. It is also possible that programs have a number of different *modes*, each with a rather narrow execution-time spectrum. WCET analysis should support the analysis of such modal behavior by allowing for the use of several different sets of input assumptions, each corresponding to the activation of a certain mode in the software (see Section 2.2.1).

1.10 Industrial Practice and Attitudes

According to a survey by Ermedahl and Gustafsson [EG97b, Gus00], WCET analysis is used in industry to verify real-time requirements, to optimize programs, to compare algorithms, and to evaluate hardware. None of the companies contacted in the survey used a commercial WCET tool (since no such tools were available), instead they used measurements or manual timing analysis to estimate worst-case times.

A wide variety of measurement tools are employed in industry, including emulators, time-accurate simulators, logic analyzers and oscilloscopes, timer readings inserted into the software, and software profiling tools [Ive98].

The reported consensus among the developers contacted in the survey was that a WCET tool would be valuable, since it would save the development time spent performing measurements, allow more frequent timing analysis, and enable what-if analysis by selecting various CPUs and processor speeds.

In the space industry, WCET tools have been available for some years, even though they have not been adapted by mainstream embedded developers [HLS00a, HLS00b]. It seems likely that the aerospace and automotive industries will be the leading industries in accepting static WCET analysis as a mainstream tool, since they build many embedded safety-critical real-time systems [FHL$^+$01].

1.11 Contributions of This Thesis

The quality of real-time software is to a large extent dependent on the quality of the analysis methods applied to it. In particular, quality execution time estimates for the software are needed to make it possible to determine the timing behavior of a system before it is deployed. Static WCET analysis is a promising technology to determine the timing behavior of programs, especially for programs used in embedded real-time systems.

Tool support is necessary to perform WCET analysis, since the calculations involved can be very complicated, especially for large programs. For a tool to be useful in real life, it has to be efficiently *retargetable* so that the many different processors used in the embedded world can be targeted with minimal effort. It also needs to be *flexible*, in the sense that different target systems require different types of analyses to be performed.

The underlying technology used needs to be reasonably *efficient*, so that development work does not have to stall waiting for WCET analysis results. The technology must also have *broad applicability*, covering all or most of the types of hardware and software used to build embedded real-time systems. Finally, the *correctness* of all analysis methods used is central to build a tool that actually produces safe estimates.

This thesis addresses the technology issues of building a broadly applicable and efficient timing model of a program, and the architectural issues of how to construct a portable, correct, and flexible WCET analysis tool. The technical contributions towards those goals are the following:

- A *tool architecture* for the modularization of WCET analysis tools is presented in Chapter 3. This architecture divides the WCET analysis task into several modules, with well defined interfaces that allow modules to be replaced independently of each other. This is intended to increase the flexibility and retargetability of a WCET tool, by reducing the amount of work required to implement new features and allowing the reuse of existing modules in new combinations. Correctness should be enhanced since it is easier to validate the functioning of isolated modules.
  The types of modules in our architecture are flow analysis (determining the possible program flows), global low-level analysis (caches, branch prediction, and other global hardware effects), local low-level analysis (pipeline analysis, generating concrete execution times for program parts), and calculation (where flow and execution times are combined to determine the overall WCET estimate). Several flow analysis and global low-level analysis modules can be used simultaneously. The interfaces are a program structure and flow description based on basic blocks annotated with flow information and information about the low-level execution of instructions (the scope graph), and concrete low-level timing information in the form of a timing model (as presented in Chapter 4).

- The low-level *timing model* presented in Chapter 4 is used communicate timing information from the low-level timing analysis to the calculation. The timing model assumes that a program is represented as a set of basic blocks, and ascribes times to these basic blocks. To account for the potential overlap of instructions executing on a pipelined processor, timing effects (values that should be added to the basic block times) are given for sequences of basic blocks. The goal is to capture all effects that a certain instruction can have on the execution of other instructions in a pipelined processor, and this means that effects across more than two instructions and basic blocks have to be considered. Usually, timing effects indicate speedups due to pipeline overlap, but sometimes, timing effects indicate that extra pipeline stalls occur, for example when two non-adjacent instructions interfere with each other.
- The *timing analysis* method presented in Chapter 6 is a (local low-level analysis) module that generates the timing model for a program. It has been designed to use existing trace-driven simulators instead of a special-purpose model for WCET analysis, in essence assuming that the hardware model is a black box. This loose coupling is intended to reduce the effort required to adapt the WCET tool to a new target processor.
  Since the information from the timing analysis is deposited in the timing model, the WCET calculation is independent of the timing analysis and hardware used. Both the timing analysis and the timing model are applicable to a broad spectrum of embedded processors, and do not constrain the structure of the programs to be analyzed (even spaghetti code is admissible).
- The *formal pipeline model* presented in Chapter 5 is used to reason about the timing behavior of pipelined processors, in particular considering the times and timing effects of our timing model. Using the model, we analyze the correctness and tightness of our timing model, and determine when timing effects can occur and when they have to be accounted for to generate safe WCET estimates. We further prove some properties of certain classes of pipelines, such as the absence of timing anomalies and the absence of interference between non-adjacent instructions, and discuss the safety of other pipeline timing analysis methods in the light of our formal model.
- Chapter 8 discusses the *construction of hardware models* for static WCET analysis tools. A WCET tool has to use some model of the processor for which it is performing WCET analysis, and getting this model correct is crucial to obtaining correct WCET estimates. Unfortunately, it is very hard to prove that a hardware model is correct visavi the hardware. We provide some advice on how to build and validate processor models, and how to select modelable hardware.
- As described in Chapter 9, we have performed *extensive experiments* to evaluate the correctness, precision, and efficiency of our timing analysis method. We have implemented a prototype tool based on the WCET tool architecture, with machine models for two embedded RISC processors, the NEC V850E and the ARM9.

1.12 Outline

The rest of this thesis is organized as follows:

- Chapter 2 gives an overview of static WCET analysis and previous work in the field.
- Chapter 3 presents the modular architecture for WCET analysis tools and gives a short overview of the interface data structures.
- Chapter 4 presents the low-level timing model and its background in the behavior of processor pipelines.
- Chapter 5 presents the formal model of processor pipelines and the proofs of several properties of pipelines relevant to WCET analysis.
- Chapter 6 presents the timing analysis method used to generate the timing model.
- Chapter 7 shows how the timing analysis is applied to the NEC V850E and ARM9 processors.
- Chapter 8 discusses the issues involved in constructing a precise processor simulator.
- Chapter 9 presents the prototype implementation used to experiment with the timing model and timing analysis, and the results of our experiments.
- Chapter 10 draws conclusions from the work presented and provides a discussion on the applicability and future of WCET analysis.

Chapter 2

WCET Overview and Previous Work

This chapter presents an overview of previous work in WCET analysis, using a conceptual map of WCET analysis as the organizing principle.

2.1 Components of WCET Analysis

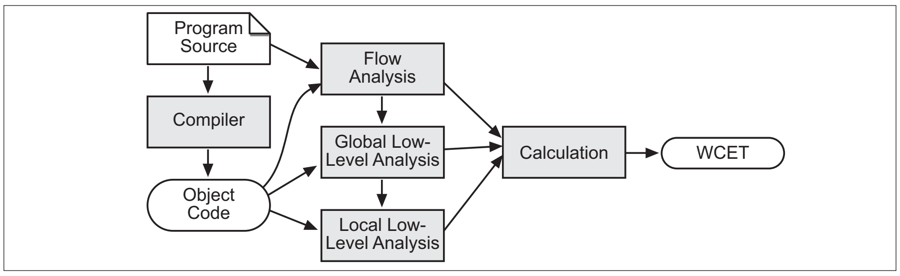


Figure 2.1: Components of WCET analysis

We divide static WCET analysis into four distinct components or phases, as illustrated in Figure 2.1.

- *Flow analysis* analyzes the program source code and/or object code, and determines the possible flows through the program.
- *Global low-level analysis* determines the effect on the program timing of global machine-dependent factors, like cache memories. The global low-level analysis might use information from the flow analys to obtain better results.
- *Local low-level analysis* determines the effect on the program timing of machine-dependent factors that can be handled locally for a few instructions, like pipelines. It might use both flow and global low-level information to obtain better results.

- *Calculation* combines the results of the three other components to find the actual WCET of the program.

This division is useful as guide to understand the field, and some variant of this conceptual map is generally used in the WCET literature. It is also the base of the tool architecture introduced in Chapter 3. Our contributions to local low-level analysis are presented in Chapters 4, 6, and 7.

2.2 Flow Analysis

The purpose of program flow analysis is to determine the possible paths through a program, i.e. the dynamic behavior of the program. The result of the flow analysis is information about which functions get called, how many times loops iterate, if there are dependences between different `if`-statements, etc. Since the problem is computationally intractable in the general case, a simpler, approximate analysis is normally performed. This analysis must yield safe path information, i.e. all feasible paths must always be covered by the approximation.

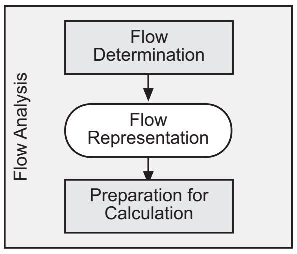


Figure 2.2: Components of flow analysis

Flow analysis can be further subdivided into three stages, as illustrated in Figure 2.2:

- *Flow Determination*, the actual analysis of the code.
- *Flow Representation*, the representation of the information obtained in the analysis phase.
- *Preparation for Calculation*, where the information is processed to be useful for the particular calculation module used.

Sometimes, it is also necessary to *map* flow information from the source-code level used in flow analysis to the object-code level used in calculation.

2.2.1 Flow Determination

The flow information can be calculated *manually*, and communicated to the WCET tool by adding annotations to the program source code or by giving additional information outside the program code.

Puschner et al. [KP01, Kir02] allow flow information to be entered into the program source code by extending the C language with additional syntax to define loop limits and path information. Börjesson [Bör95] takes a different approach by using `#pragma`s in C code instead of altering the language syntax.

Adding information outside the source code is necessary if we assume that the compiler cannot be changed to support WCET analsysis, when we want to give information about the flow without using the structure of the source code, or we want to be able to try several different scenarios for the same program without having to recompile it. Park [Par93] defines IDL (Information Description Language), based on regular expressions, to describe the possible paths, while the approaches by Li and Malik [LM95] and Theiling and Ferdinand [TF98] use linear constraints to describe the program flow on the object-code level, where the constraints are considered global for the whole program. We have defined the *flow facts language* to describe the flow of a program on the object-code level using constraints that are local to a small part of the program rather than global [EE00].

Automatic flow analysis can be used to obtain flow information from the program source code without manual intervention. The different approaches all have different complexity, generate different amounts of information, and can handle different levels of program complexity.

Altenbernd [Alt96, SA00] and Chapman et al. [CBW94] use symbolic execution to find flow information, while Ermedahl and Gustafsson [EG97a, Gus00] use abstract interpretation on the source code. Lundqvist and Stenström [LS99a] find path information using symbolic instruction-level simulation of the object code. Ferdinand et al. [FHL$^+$01] are able to detect some infeasible paths by analyzing the object code using abstract interpretation over the values of processor registers. Healy et al. [HSRW98, HW99, HSR$^+$00] use data flow analysis and special algorithms for loops (relying on induction variables) to find loop bounds and infeasible paths inside loops. Holsti et al. [HLS00b] use Presburger Arithmetic to calculate loop bounds for counted loops, analyzing programs on the object code level. The approach by Liu and Gomez [LG98] performs symbolic evaluation on a functional language to find executable paths. Ziegenbein et al. [ZWR$^+$01] identify segments of a program that only have a single feasible paths by following input-data dependences. Colin et al. [CP00] use symbolic evaluation to calculate the number of iterations in inner loops where the iteration count depends on the loop variables of outer loops.

To help automatic flow analysis in the case where input data to a program has effect on the possible program paths, the user typically has to provide some input. One method is to specify *modes* for a program, and then analyze the WCET given the assumptions specified in the mode [CBW94, WE01]. Holsti et al. [HLS00b] allows for several types of information (loop bounds, variable value bounds) to be added as annotations to help the automatic flow analysis, which also makes it possible to explore different program execution scenarios without changing the source code.

Annotation systems often allow for the program code to be annotated by more than just program flow. Hard-coded timing values for certain functions or statements can be given, excluding them from the WCET analysis. This is

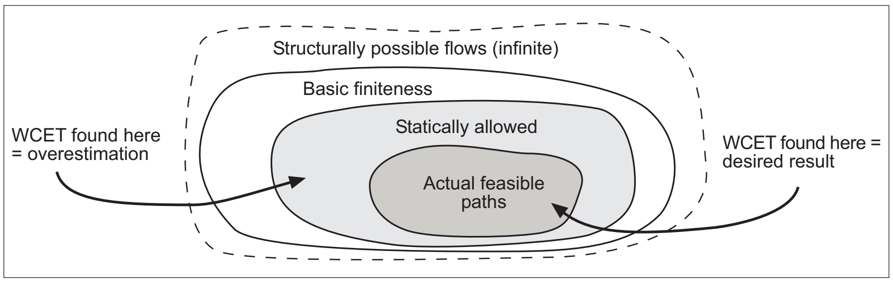

Structurally possible flows (infinite)
Basic finiteness
Statically allowed
Actual feasible paths
WCET found here = overestimation
WCET found here = desired result

Figure 2.3: Sets of possible program paths

useful for code that is not handled in WCET analysis tool (like library functions or operating systems), or to indicate the time budget for functions which are not yet implemented [HLS00b].

Note that user-provided assertions about a program must be carefully validated, and it is a distinct possibility that a static WCET estimate can be exceeded, if some assumption is violated. Obtaining good assumptions is a difficult engineering issue that is key to generating good quality WCET estimates.

2.2.2 Flow Representation

In most approaches to flow analysis, the flow representation is an intrinsic part of the flow analysis and calculation method used, but there are some approaches where flow information facts can be specified without reference to a certain calculation method or flow analysis method.

Most analysis methods generate only loop bounds, which are quite easy to separate and represent for each loop. For more complex flows, Park's IDL [Par93] and our flow facts language [EE00] can be considered explicit representations of the program flow independent of any particular analysis method and calculation method.

The use of linear constraints in some WCET analysis approaches offers a powerful representation of program flow, which is independent of the flow analysis but not of the solver [LM95, PS95, OS97, TF98, EE99].

Figure 2.3 provides an illustration of the various levels of information about the flow of a program that can be represented for use in WCET calculation. The largest set of executions of a program is given by the *structure* of the program: all paths that can be traced through the program, regardless of the semantics of the code. This set is usually infinite, since all loops (iterative or recursive) can be taken an arbitrary number of times. This set of paths is implicit in the structure of a program.

The execution of the program is made finite by introducing *basic finiteness information*, where all loops are bounded by some upper limit on the number of iterations. If such limits cannot be provided, the termination of the program must be called into question. Any real-time program should finish every loop within some bounded number of iterations (unless it is the outermost loop that keeps the system running). Basic loop bounds may not be enough to give a tight WCET estimate. In some cases, using only loop bounds gives an overestimation by a factor 100, clearly indicating the need for more advanced path modeling [PB01].

Adding even more information allows the set of executions to be narrowed down further, to a set of *statically feasible paths.* These are the paths allowed by the information we can hope to obtain about the program using offline analysis. This set will vary with the sophistication of the analysis methods used, and is hopefully coincident with the *actually feasible* paths of the program. If not, the result of the static analysis might be an overestimation of the execution time.

2.2.3 Preparation for Calculation

If a more general flow representation like our flow facts language is used, the flow information has to be preprocessed to be used by a particular calculation method. In some cases, certain information items might have to be discarded since the calculation module is unable to take advantage of them. Note that it is safe to discard facts if we assume that the flow information is used only to narrow down the set of allowed paths. Discarding facts will thus only lead to a potentially less tight WCET estimate, but not an erroneous estimate.

In [EE00], we show how to use flow facts for a constraint-based (IPET) calculation module. The same flow facts were later used in a very different calculation module based on explicit path exploration, where we could not take advantage of some classes of flow facts [SEE01b, SEE01a].

2.2.4 The Mapping Problem

There is a problem with obtaining or representing program flow based on a source code view of a program: the information has to be *mapped* down to the object-code structure of the compiled and linked program, since this is the only representational level where execution times can be generated.

Considering modern compilers, the relation between the source code and the object code is not obvious, since the compiler can perform transformations like unrolling loops, inlining functions, and duplicating code. For example, Lundqvist and Stenström [LS98] report a case where entire conditional statements were removed from the program during the compilation process. Thus, mapping flow information obtained at the source code level down to the object code level is a non-trivial task, where the transformations to the code made by the optimizations in the compiler have to be reflected in the flow information.

I attacked this problem in my Master's thesis [Eng97] by designing an external system that transforms the flow information to reflect the code transformations performed by the compiler. Kirner [KP01] uses the internal debug information propagation facilities in the `gcc` compiler to achieve the same task. Lim et al. [LKM98] proposes an approach where the compiler is assumed to maintain labels identifying relevant places in the code, allowing loops in the object code to be identified with their source code equivalents. Holsti et al. [HLS00b] map information about loop bounds and variable values by using information that allows a certain part of the code to be found based on characteristics like "the loop that modifies the variable X".

An interesting parallel is that *programming* in general has been raised to higher levels of abstraction (from assembly code to C to graphical programming tools), at the expense of much more complex programming tools. The amount of code involved in compiling a statechart diagram to executable code is much larger than an early-70s C compiler. WCET analysis will have to evolve with the programming tools: raising the level of the input abstraction for flow analysis will require a more complex mapping tool, since information has to be mapped through more layers of abstraction.

One way to avoid the mapping problem is to perform (automatic) flow analysis on the object code of a program [LS99a, HLS00a, FHL$^+$01]. However, working on the object code is unnecessarily difficult, since variables migrate between memory and registers, making it difficult to identify and track relevant data objects. Also, the code emitted by the compiler for complex constructions like switch statements and the use of complex assembly-language coding techniques in compiler libraries have to be specially recognized to be handled effectively [HLS00a].

Assuming control over a compiler, it is possible to perform the analysis *inside* the compiler on the intermediate code, which mostly avoids the mapping problem since the analysis can then be performed on the optimized program with full information about variables etc. [HW99, HSR$^+$00]. This also avoids discarding the large amount of information about a program that a compiler gathers as a part of the compilation process.

2.3 Low-Level Analysis

As mentioned, it is necessary to analyze the object code of a program to obtain the actual timing behavior. This analysis is called *low-level analysis*.

2.3.1 Global Low-Level Analysis

For some timing behaviors of a microprocessor, analysis over the whole program (or at least large parts of it) is required in order to obtain a safe and tight result. Examples of features with global effects are cache memories and branch prediction. To determine the cache behavior of an instruction, the analysis must consider many instructions, arbitarily remote from the instruction considered.

Since exact analysis is normally impossible for global features, an approximate but safe analysis is necessary. For example, when an attempt is made to determine whether a certain instruction is in the cache, a cache miss is assumed unless we can be absolutely sure of a cache hit. This might be pessimistic but is safe (assuming that the types of timing anomalies discussed in Section 2.3.2 do not occur).

Instruction caches are quite easy to analyze, since the instruction fetch behavior can be determined from the program flow. *Data caches* are more difficult, since the data access patterns of a program are hard to predict statically, and a single instruction can access many different memory locations.

Some approaches integrate the cache analysis into the calculation phase. Lim et al. [LBJ$^+$95] analyze the instruction cache behavior by traversing the syntax tree of a program, determining a cache behavior abstraction for each node. Kim et al. [KMH96] add data cache analysis to the approach, in a rather pessimistic way by assuming two cache misses for each unknown data access. Li et al. [LMW96] build a cache conflict graph to model conflicting instruction cache accesses. The graph is converted to a set of linear constraints, and solved as part of the calculation phase. Ottosson and Sjödin [OS97] also employ constraint techniques to model the execution time gains from using instruction caches and data caches. Lundqvist and Stenström [LS99a] perform instruction and data cache analysis together with flow analysis and calculation in a modified processor simulator.

A different approach is to perform a separate cache analysis phase to determine the instruction cache behavior, and then to use this information in the local low-level analysis or the calculation phase. In this case, an explicit representation of the cache analysis result is used. Ferdinand et al. [FMW97] use abstract interpretation techniques to conservatively estimate instruction cache hits and misses. Healy et al. [HAM$^+$99] perform a static cache simulation which generates a categorization of each instruction cache access as one of *always miss, always hit, first miss, first hit.* This approach has been extended to instruction cache hierarchies with several levels of cache [Mue97]. White et al. [WMH$^+$97] extends the approach to data caches, classifying instructions in a similar manner, exploiting the data locality inherent in loops over arrays to decrease the number of pessimistic miss categorizations. The instruction cache analysis used by Stappert and Altenbernd [SA00] uses data flow analysis methods to determine cache hits and misses for programs without loops (the pipeline state is determined simultaneously).

Unified caches are generally considered a very difficult problem, since data and instruction accesses go to the same cache, giving a very complex behavior. However, by integrating the pipeline and cache analysis, Ferdinand et al. [FHL$^+$01] have constructed a low-level analysis for a processor employing a unified cache, where value analysis using abstract interpretation is performed on the object code of the program in order to determine the potential target addresses of data access instructions.

Branch prediction is another global effect that can have a large effect on the execution time of a program. Colin and Puaut [CP00] model the effect of the branch target buffers (BTB) and branch prediction mechanism of the Intel Pentium processor. Mitra and Roychoudhury [MR01] model the effect of advanced branch predictors with global histories using linear constraints. The analysis is integrated with the calculation of the longest path, and can take quite long to run even for small programs.

Petters and Färber [PF99, Pet00] do not rely on a model of the system but rather run code on the actual target system, with instrumentation code inserted to control the execution and explore the longest executable path. This "models" both the caches and branch prediction of the Pentium III and Athlon processors (as well as all other processor behavior), but it is hard to guarantee the worst case. Wolf and Ernst [WE01] also use instrumentation code to guide the execution of a program, using a simulator of the hardware.

Patterson [Pat95] performs an analysis of the taken/not-taken probability of branches, for the purpose of compiler optimization rather than WCET analysis (which means that approximations do not have to be safe).

The results of global low-level analysis can be incorporated into the final WCET estimate in two different ways. The simplest approach is to assign a certain execution time penalty to bad cases like cache misses, and then add this penalty to the execution time estimate to account for the global effect [LBJ$^+$95, KMH96, TF98, CP00]. A more precise approach is to use the global results as input to the local low-level analysis, and simulate the result of the cache miss on the actual execution of instructions in the processor pipeline [HAM$^+$99, EE99, SA00, FHL$^+$01].

2.3.2 Local Low-Level Analysis

The local low-level analysis handles machine timing effects that depend on a single instruction and its immediate neighbors. Examples of such effects are *pipeline overlap* and *memory access times*. Almost all research in local low-level analysis has been directed at pipeline analysis. Just like in global analysis, approximated but safe approaches are sometimes necessary, but thanks to the simpler behavior of local effects, the precision is usually higher for local low-level analysis.

For simple processors with no pipeline or very simple pipelines, a constant execution time per instruction might be given [HLS00b], but for more complex, pipelined architectures, a more in-depth analysis is necessary to provide tight execution time bounds.

Theiling and Ferdinand [TF98] use simplified instruction timing in order to model the effect of cache missed on the execution time of instructions, assuming a fixed execution time of one cycle for all instruction when they hit the cache and 10 cycles otherwise.

As illustrated in Figure 2.4, it is possible to describe the execution time of a pipelined processor by ignoring the overlap between instructions which "always" occurs, and rather use the *interarrival time*, which is done in many CPU manuals. Compared to the *block time* in Figure 2.4, the interarrival time represents a form of "steady state" in the pipeline. Processor manuals written in this style give a number of cycles for each instruction rather than a detailed pipeline diagram, and some rules for when the execution gets slowed down due

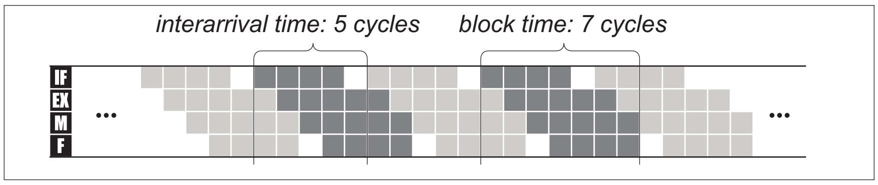


Figure 2.4: Block times and interarrival times

to resource contention [Inf01, ARM00b]. Atanassov et al. [AKP01] have built a model for the pipelined Infineon C167 processor using constant execution times for each instruction plus formulas that account for the interference between neighboring instructions, the effect of memory access times, etc. We use a similar model for the ARM9 (see Section 7.2).

Lim et al. [LBJ$^+$95] analyze the pipeline overlap between program basic blocks in conjunction with instruction cache analysis. The target processor is a MIPS R3000/3010, with parallel integer and floating-point pipelines and in-order issue, modeled using reservation tables similar to our pipeline diagrams. The approach is extended to in-order superscalar processors, by maintaining instruction dependence and latency graphs instead of reservation tables. The target is a fictive superscalar processor [LHKM98].

Healy et al. [HAM$^+$99] analyze the concatenation of reservation tables for a MicroSPARC 1 processor (similar in complexity to the MIPS R3000/3010), over paths inside loops and functions. They are able to capture pipeline effects affecting more than just neighboring basic blocks.

Ottosson and Sjödin [OS97] express the effect of pipeline overlap between basic blocks as negative times. The model presented in this thesis is a generalization of this timing model, allowing for overlaps between blocks which are not neighbors, and presenting a method to obtain the execution times for a particular program and processor.

Colin and Puaut [CP01a, CP01c] perform WCET analysis for one of the integer pipelines of a Pentium processor, using reservation tables.

The analysis of Altenbernd and Stappert [SA00] performs pipeline analysis for a somewhat simplified superscalar PowerPC 604. The pipeline analysis is integrated with the instruction cache analysis and computes a pipeline reservation table for each basic block. These tables are then concatenated in the calculation phase to find the longest path.

Wolf and Ernst [WE01] analyze the pipeline behavior of the scalar StrongARM and SPARClite processors using processor simulators, or special development hardware for the processors, avoiding the need for a special pipeline model.

Schneider and Ferdinand [SF99] analyze the in-order superscalar SuperSPARC I using an abstract pipeline state to model the timing effect of the superscalar pipeline, assuming that cache hit and miss information is available. Ferdinand et al. [FHL$^+$01] analyze the pipeline and cache for the scalar ColdFire 5307 processor, also using an abstract pipeline model.

Bate et al. [BBMP00] form a WCET model for a Java Virtual Machine (JVM) by calculating an execution time for each JVM instruction in isolation, and a speedup for each pair of JVM instructions when executed in sequence. The assumption is that a very simple ahead-of-time compiler is used to translate Java byte codes to native code for the target platform.

Lundqvist and Stenström [LS99a] perform pipeline analysis for a simplified PowerPC processor with in-order dual instruction issue, as part of their instruction-level simulation, integrated with the cache and flow analysis. They also noted [LS99b] that out-of-order processors are subject to *timing anomalies* that potentially makes the analysis of such processors inachievably complex. The problem is that, for example, assuming a cache hit for a certain instruction can lead to an overall greater execution time than assuming a cache miss (the commonly assumed local worst case for cache analysis). Thus, an analysis would in principle have to investigate all possible execution paths, instruction reorderings, and cache hit/miss scenarios for the entire program to be sure to find the worst-case execution time[1].

Petters and Färber [PF99, Pet00] perform the pipeline analysis for the superscalar Pentium III and Athlon processors by running code on the hardware. The analysis is thus intrinsically integrated with the global low-level analysis.

Burns and Edgar [BE00] have performed experiments to demonstrate the unpredictability of execution times on a superscalar Pentium II processor, and concluded that statistical models must probably be used to model such processors. Their assumption is that a modern processor can be considered as a set of independent units affecting the execution time, allowing for statistical analysis of the execution time variations [EB01].

Burns et al. [BKY98] describe how to model a fictive superscalar processor using Colored Petri Nets, but does not use the model for WCET analysis, only as a model to simulate the processor timing.

2.4 Calculation

The purpose of the calculation is to calculate the WCET estimate for the program, given the program flow and global and local low-level analysis results. There are three main categories of calculation methods proposed in the literature: *path-based*, *tree-based*, or *IPET (Implicit Path Enumeration Technique).*

2.4.1 Tree-Based

In a tree-based calculation the WCET estimate is generated by a bottom-up traversal of a tree corresponding to a syntactical parse tree of the program,

[1]All WCET analysis methods for caches presented in the literature assume that a cache miss is worse than a cache hit, which makes it possible to assume a cache miss in cases of uncertainty. For an out-of-order superscalar, this assumption cannot be used, and hence all possibilities for all uncertain instructions must be investigated, giving an explosion in the analysis complexity.

using rules defined for each type of compound program statement to determine the WCET of the statement. The method is conceptually simple and computationally quite cheap, but has problems handling flow information, since the computations are local within a single program statement and thus cannot consider dependences between statements.

Park and Shaw dubbed the approach “timing schema” [PS90], which is extended to include pipeline and cache states by Lim et al. [LBJ$^+$95, LHKM98, KMH96]. Chapman [Cha95] also explored the use of timing schema. Puschner et al. [PK89, PPVZ92] uses a “program timing tree” which contains some path information. Bate et al. [BBMP00] perform a tree-based calculation where the number of executions of each type of JVM instruction is propagated, and not actual execution times.

2.4.2 Path-Based

In a path-based calculation, the WCET estimate is generated by calculating times for different paths in a program, searching for the path with the longest execution time. The defining feature is that possible execution paths are *explicitly* represented or determined. The path-based approach is natural within a single loop iteration, but has problems with flow information stretching across loop-nesting levels.

Healy et al. [HAM$^+$99] look for longest paths inside loops and functions, one loop nesting level at a time, with some special-case handling for cache and pipeline effects between loop levels. Flow information can be used to prune the set of allowable paths, by limiting the number of times a particular path can be executed [HW99]. Stappert and Altenbernd [SA00] investigate the longest paths in a non-looping program, and select the longest path which is also feasible. Stappert, Ermedahl, and Engblom [SEE01b] find the longest executable path for each loop nesting level, allowing for flow information to limit the set of paths explored.

Petters and Färber [PF99], and Lundqvist and Stenström [LS99a] integrate the determination of the longest path with global and local low-level analysis.

2.4.3 IPET

IPET calculation express program flow and execution times using algebraic and/or logical constraints. Each basic block and program flow edge in the program is given a time variable (t_{entity}), denoting the execution time of that block or edge, and a count variable (x_{entity}), denoting the number of times that block or edge is executed. The WCET is found by maximizing the sum $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows. The result is a worst-case count for each node and edge, and not an explicit path like in path-based calculation.

The term “implicit path enumeration” (IPET) was coined by Malik et al. [LM95]. Puschner et al. [PS95] showed how some path information could be expressed by numerical constraints, work which was later extended by our group [EE00]. Ferdinand et al. [FMW97] partially unroll loops in the program before calculation in order to increase the precision when analyzing caches.

Usually, IPET is applied across a whole program, but Holsti et al. [HLS00b]use integer linear programming on a per-function basis. Wolf and Ernst [WE01] use IPET with a slightly larger basic unit: program segments potentially containing several basic blocks, which makes their approach a hybrid between IPET and path-based calculation.

IPET constraint systems can be solved using either constraint-solving [OS97, EE99] or integer linear programming techniques (ILP) [PS95, FMW97, HLS00b], with ILP being the most popular thanks to the availability of efficient solvers. Constraint solving allows for more complex constraints to be expressed, with the corresponding risk of higher solution times.

2.4.4 Parametrized WCET Calculation

Sometimes, the input data to a program cannot be bounded in a reasonable fashion, but the data will become known when the program is invoked. For example, a sorting function in a library will have the data size fixed when called. In such cases, it is possible that instead of generating one hard WCET estimate, the result of the WCET calculation is a *formula* containing unknowns that are fixed when the WCET estimate is needed in a running system.

Chapman [Cha95] generates a symbolic time expression for a program using a tree-based calculation. Puschner and Bernat [PB01] also work within a tree-based method, generating a formula that allows for some of the path information detailed in [PK89] to be used in making the final symbolic expression tighter. Vivancos et al. [VHMW01] perform parametric WCET analysis within a path-based calculation framework, demonstrating how the timing effects of caches and pipelines can be accounted for in a potentially pessimistic and but safe manner.

Chapter 3

WCET Tool Architecture

In order to build a tool which is powerful and portable, we have defined a modular WCET tool architecture, shown in Figure 3.1. The archicture is based on well-defined data structures which provide an interface between replaceable modules. Compared to the conceptual view in Figure 2.1, interface data structures have been added and the low-level analysis has been given more detail.

The purpose of the architecture is to keep analysis modules separate and isolated, to facilitate the porting of WCET tools to new target architectures and the replacement of modules. Furthermore, clear separation between modules in combination with well-defined interfaces makes it easier to validate a WCET tool, since module-wise testing is facilitated.

There are three interface data structures in the architecture: the *object code*, the *scope graph* and the *timing model*. The object code is target-dependent by nature, and contains the compiled and linked code of the program being analyzed. The scope graph reflects the structure of function calls and loops in the program, and is used to express the results of flow analysis and global low-level analysis. The timing model communicates the timing of the program from the low-level analysis to the calculation. Within the pipeline analysis, the *timing graph* is used to represent the target program.

It is possible to string several different flow analysis or global low-level analysis methods together, each adding information to the scope graph (and potentially rewriting its structure). For example, cache analysis and branch prediction analysis modules could both be used, and the joint result represented in the scope graph. This makes it easy to extend the tool with new analyses, without disturbing the existing modules.

The modules that we consider suitable to vary are the simulator (to adapt to a new target architecture), the flow analysis (to use different analysis methods with varying precision and computational cost), the global low-level analysis (to analyze different types of caches or other factors), and the calculation (to trade speed against precision). The pipeline analysis method we prefer is the one presented in this thesis, with the simulator used to vary the target architecture.

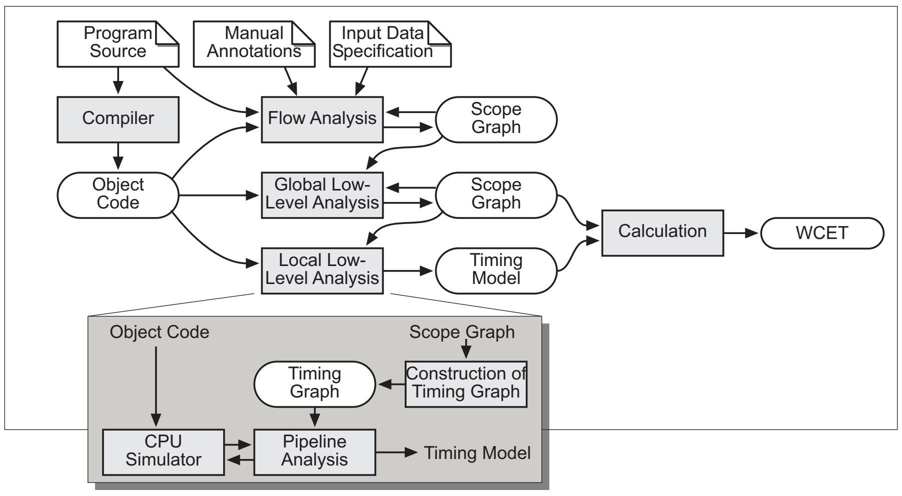


Figure 3.1: WCET tool architecture

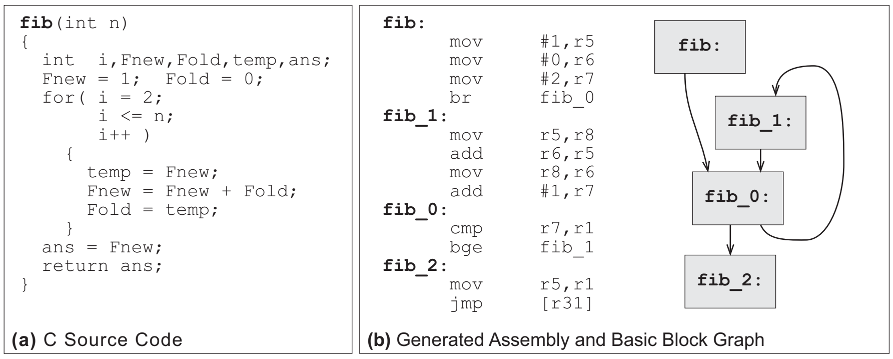
3.1 Separation vs. Integration

Our architecture is based on a clear *separation* of all analysis modules. At the other end of the design spectrum is the complete *integration* of all analysis modules into a single pass that performs both flow analysis, low-level analysis, and calculation (the best example of this is the work of Lundqvist and Stenström [LS99a]).

By clearly separating the modules, porting to new target platforms is facilitated, since only minor parts of the tool have to be replaced or rewritten. It is easier to slot in new analyses, since they can work with the defined interface data structures [EES01a, CP01a, FHL$^+$01]. This is the guiding principle behind the clear separation of *timing modeling* and *timing analysis* made in this thesis, where the timing model is the interface between the calculation and the low-level analysis.

Considering the validation of WCET tools, separating WCET analysis into modules allows for module-wise testing, which makes it easier to validate and test a tool. To guarantee that a WCET estimate produced by a WCET analysis tool is safe and tight, each analysis phase must be safe and tight in its own right. Otherwise, errors in one module could mask errors in other modules [ESE00, EES01b].

Another argument for separation is that the integration of some analyses may cause excessive solution times due to the complexity of the integrated problem, even if each separate problem is quite easy to solve. Especially vulnerable is the simultaneous analysis of caching behavior (global low-level analysis) and pipeline behavior (local low-level analysis). Solution times even for quite small programs



```
fib(int n)
{
  int  i,Fnew,Fold,temp,ans;
  Fnew = 1;  Fold = 0;
  for( i = 2;
       i <= n;
       i++ )
    {
      temp = Fnew;
      Fnew = Fnew + Fold;
      Fold = temp;
    }
  ans = Fnew;
  return ans;
}
```

(a) C Source Code

```
fib:
        mov     #1,r5
        mov     #0,r6
        mov     #2,r7
        br      fib_0
fib_1:
        mov     r5,r8
        add     r6,r5
        mov     r8,r6
        add     #1,r7
fib_0:
        cmp     r7,r1
        bge     fib_1
fib_2:
        mov     r5,r1
        jmp     [r31]
```

(b) Generated Assembly and Basic Block Graph

Figure 3.2: Example of basic block structure

can easily explode into several minutes or even hours [LMW96, OS97, FHL$^+$01]. Separating each algorithm into a module of its own also makes that module simpler, which makes implementation easier.

However, it is possible that some precision is lost when enforcing separation of modules in the case that there are circular dependences between different analyses. A good example is the analysis of unified caches[1], where the pipeline behavior affects the cache behavior and vice versa, making an integration of the two phases very attractive precisionwise [FHL$^+$01]. Another example is the use of speculative execution in processors, where instructions that are fetched and executed due to control speculation, but then discarded can affect the caching behavior of a program.

3.2 Basic Blocks

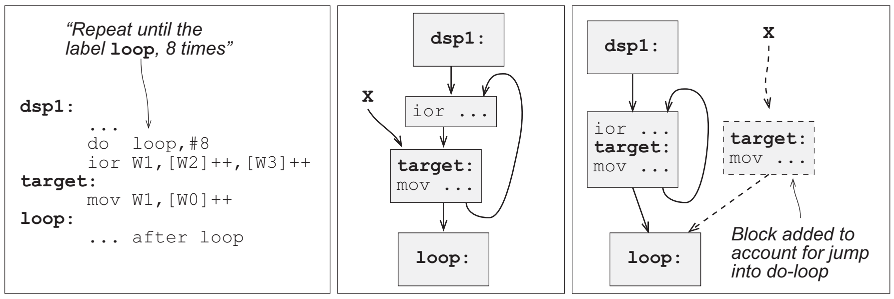
All data structures in our tool architecture are based on dividing the object code of the program to be analyzed into *basic blocks*. A basic block is a piece of sequential code that does not contain any flows into or out of it, except at the start and end [ASU86, Muc97]. Figure 3.2 shows an example of how a small C function containing a loop is compiled to object code, and the corresponding basic block graph. It is possible to use compiled code or handwritten assembly with our techniques, since it is neutral regarding how the code is generated (the use of assembly is not illustrated in Figure 3.1).

The edges in the basic block graph represent the potential flows in a program. For example, in Figure 3.2, there is an arrow from the block `fib` to `fib_0` since there is an unconditinal jump at the end of block `fib`. From `fib_0` it is possible to get to both `fib_1` if the conditional branch is taken, and to `fib_2` if the branch is not taken (fall-through). Note that not all flows corresponding to edges in the basic block graph are necessarily possible in any actual execution of the program (see the discussion about possible execution paths in Section 2.2.2).

[1]Caches where instructions and data are mixed.

Delayed branches, as employed on the SPARC and some other RISC architectures, are handled by including the instructions in the delay slot (immediately following the branch instruction) in the same basic block as the branch, which means that the last instruction of the basic block is not a branch.

The basic block model presented so far is the one used in compilers. Each machine instruction in the program is only present once in the basic block graph. However, there might be cases where the basic block model used by the WCET technique is different from the one generated by a compiler.


"Repeat until the label loop, 8 times"
dsp1:
X
ior ...
target:
mov ...
loop:
ior ...
target:
mov ...
Block added to account for jump into do-loop

```
dsp1:
      ...
      do  loop,#8
      ior W1,[W2]++,[W3]++
target:
      mov W1,[W0]++
loop:
      ... after loop
```

(a) DSP assembly **(b)** Normal basic blocks **(c)** WCET basic blocks

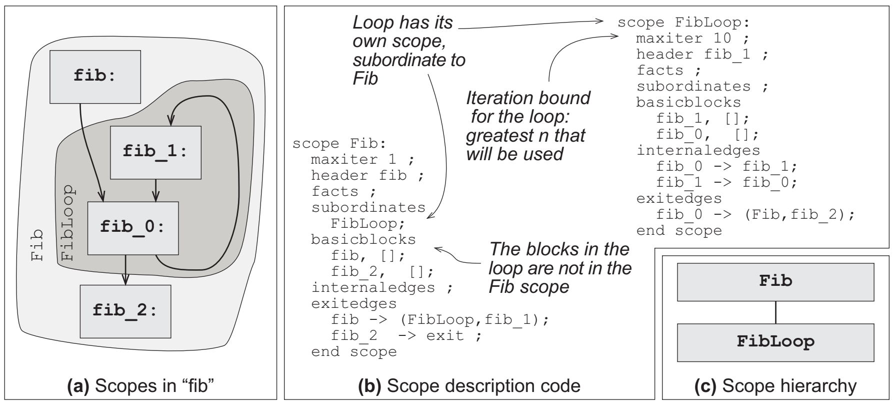
Figure 3.3: Example of an implicit jump

For example, many DSPs have instructions with semantics like "repeat the next n instructions m times". There is no jump at the end of the loop, and thus jumping into the tail of a loop will not cause a loop. Figure 3.3(a) shows an example of such a DSP-style loop. Figure 3.3(b) shows the simple basic block model, as a compiler would do it, while Figure 3.3(c) shows how the program should be modeled in a WCET tool for maximum accuracy [HLS00b]. Note the difference in where a jump from `X` enters the loop.

3.3 Scope Graph

The scope graph provides the WCET tool with the structure of the program to analyze. It is a directed acyclic graph of *scopes*. Each scope corresponds to a program feature like a loop or a function; the exact scope structure of a program depends on the analysis modules used for flow analysis and global low-level analysis. In unstructured code, loops could be identified using techniques like DJ graphs [SGL96, Ram00], but we do not prescribe any particular method. Scopes are necessary in order to carry program flow execution, in particular bounds for all loops and context-sensitive flow information for function calls.

In our current implementation (see Chapter 9), we let each function call and each loop nesting level generate a scope, thus obtaining a program structure similar to that used by Healy et al. [HAM$^+$99]. The global low-level analysis can generate new scopes in order to model the first iteration of a loop differently from subsequent iterations, in the manner pioneered by Ferdinand et al. [FMW97].


Fib
FibLoop
fib:
fib_1:
fib_0:
fib_2:
(a) Scopes in "fib"
Loop has its own scope, subordinate to Fib
Iteration bound for the loop: greatest n that will be used
The blocks in the loop are not in the Fib scope
(b) Scope description code
Fib
FibLoop
(c) Scope hierarchy

```
scope Fib:
  maxiter 1 ;
  header fib ;
  facts ;
  subordinates
    FibLoop;
  basicblocks
    fib, [];
    fib_2,  [];
  internaledges ;
  exitedges
    fib -> (FibLoop,fib_1);
    fib_2  -> exit ;
  end scope
```

```
scope FibLoop:
  maxiter 10 ;
  header fib_1 ;
  facts ;
  subordinates ;
  basicblocks
    fib_1, [];
    fib_0,  [];
  internaledges
    fib_0 -> fib_1;
    fib_1 -> fib_0;
  exitedges
    fib_0 -> (Fib,fib_2);
  end scope
```

Figure 3.4: Scope graph for the code in Figure 3.2

We have defined a textual format for *scope graph files.* The scope structure and text format of the example program from Figure 3.2 is shown in Figure 3.4. Figure 3.4(a) shows how the scope graph divides the basic block graph into a function `Fib` and a loop `FibLoop` within the function, and how the basic blocks in the program are divided between the two scopes. Figure 3.4(b) shows the textual description for the two scopes, and Figure 3.4(c) shows the scope hierarchy, where the basic blocks contained in each scope are hidden, to make the structure of the scopes relative to each other easier to see.

Each scope contains a code section that describes the code of the scope. Currently, the code section references the basic blocks from the object code of the program that are part of the scope. Extra information about the execution of each node can be added, using *execution scenarios* as described below.

Note that each basic block can be present in several copies in the scope graph. The most common example is that a function is called at several different places in the program. In such cases, each call will be described by a separate scope, and thus the code in the function will be present multiple times. Cache analysis can also give rise to multiple copies of a basic block, by unrolling a loop to differentiate between the first and successive iterations.

A separate section describes the possible flow between the basic blocks in the scope, as well as into and out of the scope. This corresponds to a subset of the flow edges of the program, as can be seen in the `internaledges` and `exitedges` sections of the scopes in Figure 3.4(b).

One of the basic blocks in each scope is designated as the *header node*, which is used to define the iteration of the scope. Usually, the header node is the first node encountered when entering the scope. A basic loop bound has to be given for each scope, and this bound corresponds to the number of times the header node can be executed for each entry into the scope. The limitation to a single header node will be removed in a future revision of the scope graph format, in order to support more complex program structures. Note that these limitations

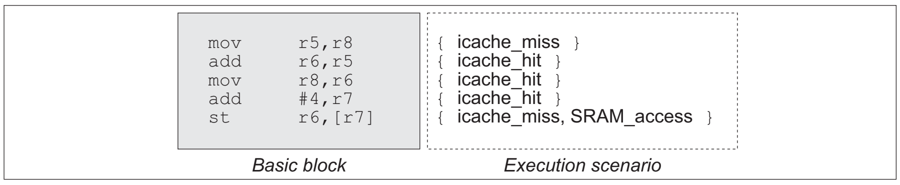

```
mov      r5,r8        { icache_miss }
add      r6,r5        { icache_hit }
mov      r8,r6        { icache_hit }
add      #4,r7        { icache_hit }
st       r6,[r7]      { icache_miss, SRAM_access }
```

Figure 3.5: Example execution scenario

in the program structure are related to the expression of program flow and thus only affects the calculation module and flow analysis module; the low-level timing analysis does not depend on any particular program structure.

Finally, more detailed flow information can be added to the scopes using the *Flow Facts Language*, which is a rather powerful language to express program flow [EE00, SEE01b]. The flow facts are independent of the calculation module and flow analysis module used.

3.3.1 Execution Scenarios

In order to perform WCET analysis for a piece of object code, we need information about factors affecting the hardware execution time for instructions that cannot be determined from the instructions themselves. Examples of such information are cache hits and misses, the speed of memory being accessed, and bound on the execution time of varying-length instructions. This information needs to be communicated from the generating analysis module to the pipeline analysis module.

Our solution to this problem is to attach *execution scenarios* to the basic blocks in the scope graph. Every instruction in a basic block can have zero or more *execution facts* attached, providing information about the execution of that instruction. The execution facts for all the instructions in a basic block form an execution scenario. Figure 3.5 shows an example of an execution scenario for a basic block, with information from memory access analysis and cache analysis.

The information in the execution scenarios is used by the pipeline analysis when determining the execution time of the basic block. It is possible to have several execution scenarios for each basic block in a program, each represented in a separate scope.

Each execution fact has to have a constant effect on the execution of the associated instruction. Cache behavior characterizations like `first-miss` and `first-hit` proposed by Healy et al. [HAM+99] are not valid as execution facts. In this case, two execution scenarios would have to be set up, one with a hit and one with a miss, and the scope graph modified to include a basic block for each alternative, with flow information added to make the WCET calculation count each node an appropriate number of times.

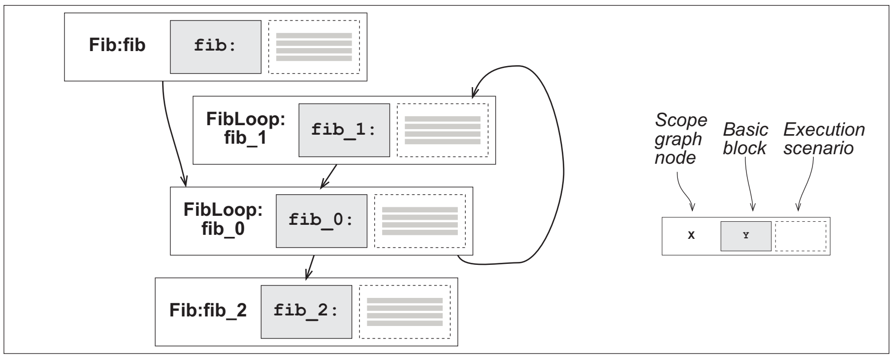


Figure 3.6: Timing graph for the program in Figure 3.2 and Figure 3.4

3.4 Timing Graph

The pipeline analysis is performed using a graph of basic blocks with execution scenarios called the *timing graph*, which is generated from the scope graph (as shown in Figure 3.1). The timing graph is a flat representation of the program, and the hierarchy from the scope graph is not present. As explained in Section 3.3, an individual basic block in the program can be present many times in the analysis for a program, with a different execution scenario associated with each occurrence (in a practical implementation, the timing graph nodes simply contain pointers to the basic blocks and the execution scenarios, no data needs to be copied).

The times generated from the timing graph have to be related back to the scope graph, since the calculation module has to combine the program flow information expressed in the scope graph with the times from the timing analysis. Thus, each node in the timing graph has to be labeled with the scope graph node it is generated from.

Figure 3.6 shows the timing graph for the `Fib` program, with scope graph correspondences, basic blocks, and execution scenarios.

3.4.1 Global or Local Timing Graphs

The normal case is to build a timing graph and perform timing analysis across the *entire program*. Doing a global timing analysis makes sure that all timing effects are found, regardless of their relation to any boundaries used in the calculation phase. The results of the analysis can then be used in their entirety in a global calculation method like IPET, or have pieces extracted to be used in the local calculations typical of path-based approaches [SEE01a].

The analysis can also be performed within smaller units of a program. This could be appropriate, for example, inside a compiler employing separate compilation. The functions in each file would be analyzed locally, from which a global result can be generated in a later, global, pass. Note that in this case, the calculation module has to employ some safe approximation for timing effects across file boundaries, since no information about such effects can be generated.

Chapter 4

Timing Model

This chapter presents the *timing model* used in our WCET tool architecture. The timing model is an abstract *representation* designed to capture the execution time of a program running on some particular hardware, in such a way that the program path with the longest total execution time can be efficiently found. In particular, we are interested in modeling the timing effects of a processor pipeline, and will sometimes call the timing model a *pipeline timing model.*

The *analysis* used to obtain the timing model for a program is described in Chapter 6. This separation of analysis and representation is shown in Figure 4.1, illustrating the important point that the hardware model (processor simulator) employed is isolated from the program timing model and the calculation. The basis of the model is described intuitively in Section 4.3, and formally in Chapter 5.

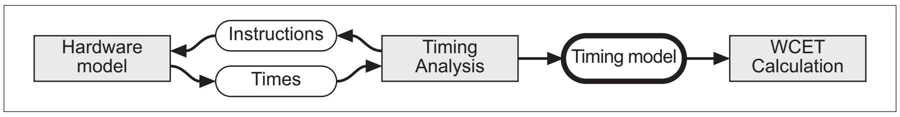


Figure 4.1: Isolating the timing model from the hardware model

The goal of our pipeline timing model is to allow the execution time of a program to be composed from smaller parts. This avoids the need to analyze the timing of complete execution paths in the program, thus increasing the efficiency of the WCET analysis tool, since the number of paths can potentially be very large. To support the use of a hardware model that is treated as a black box, we store only concrete execution times in the model, and not some state related to a special-purpose pipeline model.

The timing model is based on the timing graph, and can be seen as a decoration of the graph with times for nodes and sequences of nodes.

4.1 The Hardware Model

We consider the hardware model used in the timing analysis and timing model to be a function T that returns an execution time given a sequence of nodes in the timing graph (containing instructions and execution facts). We use the notation $T(N_1 \dots N_n)$ for the execution time of a sequence of nodes $N_1 \dots N_n$. For simplicity, we define the execution time of the empty sequence as zero, i.e. $T() = 0$.

We assume that the hardware is deterministic: each time a certain sequence of instructions or nodes is executed from the same initial condition, the same execution time results. Each node, or sequence of nodes, in the timing graph has a single execution time. All effects of variable factors like caches are handled by using execution facts to fix the behavior for the instructions in a certain timing graph node. To capture some global low-level effects, a node might have to be split into several nodes with different execution facts.

In general, the times in the timing model may be overestimates of the actual execution time, since this might be necessary to obtain safe WCET estimates. For example, an instruction with unknown caching behavior can be classified as a cache miss, and the cache miss time used to calculate the execution time for that instruction in the timing model. Instructions with variable execution time due to input data will have to assume the worst-case execution, unless we can determine the possible values of the input data.

The timing model is independent of how the timing function T defines the execution time of a piece of code. The exact definition of the execution time reported by T depends on the hardware model. The function T is simply an oracle that assigns some time to each sequence of nodes, in a deterministic manner, and we do not know how this oracle determines its execution time estimates. For a typical model of a pipelined processor, the time will be from the time the first instruction enters the pipeline until the last instruction exits the pipeline, as illustrated by the "block time" in Figure 2.4. With this definition of execution time, for two nodes `A` and `B` executing in sequence on a pipelined processor, typically $T(\texttt{AB}) < T(\texttt{A}) + T(\texttt{B})$, thanks to the pipeline overlap between the two nodes.

In principle, the weak requirements on the hardware model allow us to use any *trace-driven* cycle-accurate processor model (or even the hardware itself) to perform WCET analysis, making it much easier to retarget the WCET analysis to new processors. A trace-driven simulator does not need to simulate the semantics of the code executed (like the results of arithmetic operations or the contents of memory locations), it need only provide execution times for a given sequence of instructions. This type of simulators is commonly used in microprocessor design to evaluate new features and design tradeoffs [BC98].

Ziegenbein et al. also use simulators or hardware to obtain execution times for program segments, but in contrast to our approach, they use simulators that include the code semantics, making it more difficult to guide the execution to particular parts of the code [ZWR$^+$01].

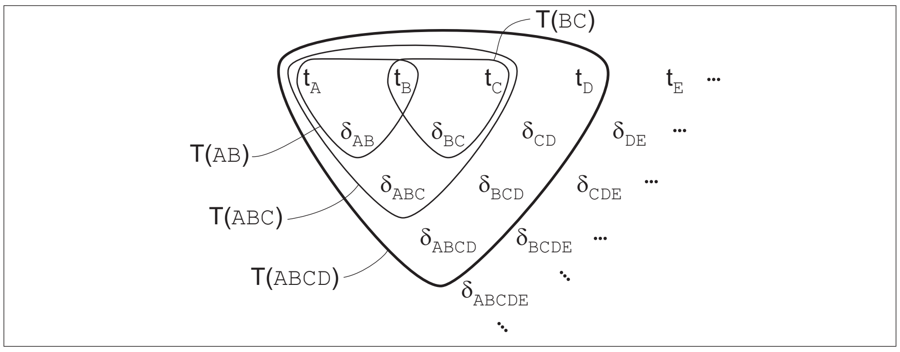


Figure 4.2: Times and the timing model

4.2 The Timing Model

The timing model is a way to represent the times reported by the timing function T, without storing the times for all possible sequences of nodes in the timing graph. Instead, we represent the times using *node times*, denoted by t_N, and *timing effects* for sequences of nodes, denoted by $\delta_{N_1 \dots N_n}$. The definitions for t and δ are the following:

$$t_N = T(N) \tag{4.1}$$

$$\delta_{N_1 \dots N_n, n \geq 2} = T(N_1 \dots N_n) - T(N_2 \dots N_n) - T(N_1 \dots N_{n-1}) + T(N_2 \dots N_{n-1}) \tag{4.2}$$

The execution time for an arbitrary sequence of nodes in the timing graph can be obtained by adding the node times for all the nodes in the sequence and the timing effects of all subsequences (of length ≥ 2) of the sequence, which should give the same time as if the sequence was run in the timing function, as shown in Equation (4.3). The δ values are added to the basic execution times, and thus they are negative in the case of a pipeline overlap. Figure 4.2 illustrates how times for successively longer sequences are constructed from Equation (4.3): the execution time $T(N_1 \dots N_n)$ for some sequence $N_1 \dots N_n$ is obtained by summing all the t and δ variables within the corresponding triangle.

$$T(N_1 \dots N_n) = \sum_{j=1}^{n} t_{N_j} + \sum_{1 \leq i < k \leq n} \delta_{N_i \dots N_k} \tag{4.3}$$

Note that we are only interested in the sequences of nodes that can actually occur in the program; that is only sequences where the successive nodes are linked by edges in the timing graph.

It is natural to make a distinction between *pairwise timing effect*, that is, effects over neighboring nodes ($N_1 N_2$), and *long timing effects*, timing effects across three nodes or more. The pairwise timing effects are expected in any pipeline (as shown in Section 4.3.1), while long timing effects are less common and more complex to find and handle (as discussed in Section 4.3.2 and Chapter 5). *Positive timing effects* add execution time to a program, and are critical to consider since otherwise an underestimate of the WCET could result. *Negative timing effects* indicate potential savings in execution time, and ignoring them only makes the WCET estimate less tight.

For an example of how Equation (4.3) works, consider the case of a sequence of two nodes, N_1 and N_2 (note that the empty sequence takes zero time):

$$\begin{aligned} t_{N_1} &= T(N_1) \\ t_{N_2} &= T(N_2) \\ \delta_{N_1 N_2} &= T(N_1 N_2) - T(N_1) - T(N_2) + T() \\ &= T(N_1 N_2) - t_{N_1} - t_{N_2} \\ T(N_1 N_2) &= t_{N_1} + t_{N_2} + \delta_{N_1 N_2} \end{aligned}$$

Extending the example with a third node, N_3, we get the following:

$$\begin{aligned} \delta_{N_1 N_2 N_3} &= T(N_1 N_2 N_3) - T(N_1 N_2) - T(N_2 N_3) + T(N_2) \\ T(N_1 N_2 N_3) &= t_{N_1} + t_{N_2} + t_{N_3} + \delta_{N_1 N_2} + \delta_{N_2 N_3} + \delta_{N_1 N_2 N_3} \end{aligned}$$

Considering a certain complete execution of a program, represented as a (very) long sequence of nodes $N_1 \ldots N_n$, we get a very large number of times and timing effects. In theory, we need the timing effect for every sequence of nodes that is a subsequence of $N_1 \ldots N_n$ (and of length ≥ 2), which is a rather large set (it contains $N + (N-1) + \ldots + 1 = N(N+1)$ terms):

$$\begin{aligned} T(N_1 N_2 N_3 \ldots N_n) = {} & t_{N_1} + t_{N_2} + t_{N_3} + \cdots + t_{N_{n-1}} + t_{N_n} \\ & + \delta_{N_1 N_2} + \delta_{N_2 N_3} + \cdots + \delta_{N_{n-1} N_n} \\ & + \delta_{N_1 N_2 N_3} + \cdots + \delta_{N_{n-2} N_{n-1} N_n} \\ & + \vdots \\ & + \delta_{N_1 N_2 \ldots N_{n-1} N_n} \end{aligned}$$

This is obviously very inefficient, but typically, most of the timing effects will be zero. In Chapter 5 we will discuss when non-zero timing effects appear and how we can bound them, and in in Chapter 6 we will give a practical algorithm to build the timing model for a program.

To help us understand the source of the timing effects, we define a new entity, $\epsilon_{N_1 \ldots N_{n-1} | N_n}$, which corresponds to the change in total execution time when the sequence $N_1 \ldots N_{n-1}$ is extended by N_n (the case $n = 1$, when there is no sequence to start with, is defined as zero):

$$\begin{aligned} \epsilon_{N_1 \ldots N_{n-1} | N_n} &= T(N_1 \ldots N_n) - T(N_1 \ldots N_{n-1}) - T(N_n) && (4.4) \\ \epsilon_{| N_1} &= 0 && (4.5) \\ T(N_1 \ldots N_n) &= T(N_1 \ldots N_{n-1}) + T(N_n) + \epsilon_{N_1 \ldots N_{n-1} | N_n} && (4.6) \end{aligned}$$

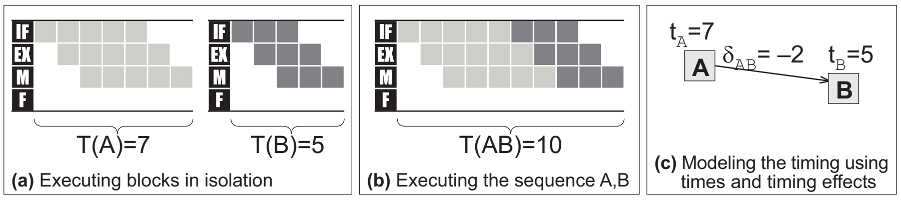

IF
EX
M
F
T(A)=7
T(B)=5
(a) Executing blocks in isolation
T(AB)=10
(b) Executing the sequence A,B
t_A=7
A
δ_{AB}= –2
t_B=5
B
(c) Modeling the timing using times and timing effects

Figure 4.3: Pipeline overlap between nodes and its timing model

Proposition 4.1 $\delta_{N_1 \ldots N_n} = 0 \Longleftrightarrow \epsilon_{N_1 \ldots N_{n-1}|N_n} = \epsilon_{N_2 \ldots N_{n-1}|N_n}$

Proof: From Equation (4.4) together with Equation (4.2), we get

$$\delta_{N_1 \ldots N_n} = \epsilon_{N_1 \ldots N_{n-1}|N_n} - \epsilon_{N_2 \ldots N_{n-1}|N_n}$$

Proposition 4.1 follows immediately. □

From this, we can see that a timing effect $\delta_{N_1 \ldots N_n}$ occurs for a sequence of nodes $N_1 \ldots N_n$ when the execution of nodes $N_2 \ldots N_n$ in the sequence $N_1 \ldots N_n$ is different from the execution of the nodes $N_2 \ldots N_n$ starting with node N_2. This is the case when N_1 has some effect that can propagate all the way to N_n. A number of examples will be shown in the next section.

4.3 Hardware Background

In the following, we will illustrate how timing effects relate to the actual execution of instructions on some pipelined processors. The goal is to give an intuitive understanding for the relation between the timing model and the behavior on the hardware. Times are measured from the point in time where the first instruction enters the pipeline until the point where the last instruction exits the pipeline.

4.3.1 Pairwise Timing Effects

Figure 4.3 illustrates the simple case for a pair of nodes where $T(\texttt{AB}) < T(\texttt{A}) + T(\texttt{B})$. The overlap between the two timing graph nodes `A` and `B` is modeled by a negative timing effect over `AB`. When executed in isolation, `A` has an execution time of 7 cycles, and `B` of 5 cycles. However, the path `AB` only takes 10 cycles, since `A` and `B` partially overlap. This makes the timing effect over the sequence `AB` minus two cycles: $\delta_{\texttt{AB}} = -2$.

It should be noted that the state of the pipeline when executing node `B` following node `A` is different from the initial state used when executing `B` in isolation. However, the effect this has on the execution time of node `B` is included in the timing effect over `AB`, since it is caused by node `A`.

Figure 4.3 also illustrates that our model can handle the same hardware timing effects as approaches using reservation tables to explicitly model the pipeline behavior [LBJ+95, HAM+99, SA00, CP01c]. In such models, each basic block in the program is modeled by a reservation table similar to our pipeline diagrams. These tables are pushed together as far as possible to model the pipeline overlap between nodes. Thus, the timing resulting from a reservation table model can be expressed in our model by simply using the length of the reservation table for a node as the time of the node. The timing effect between two nodes corresponds to how far two tables for successive nodes can be pushed together.

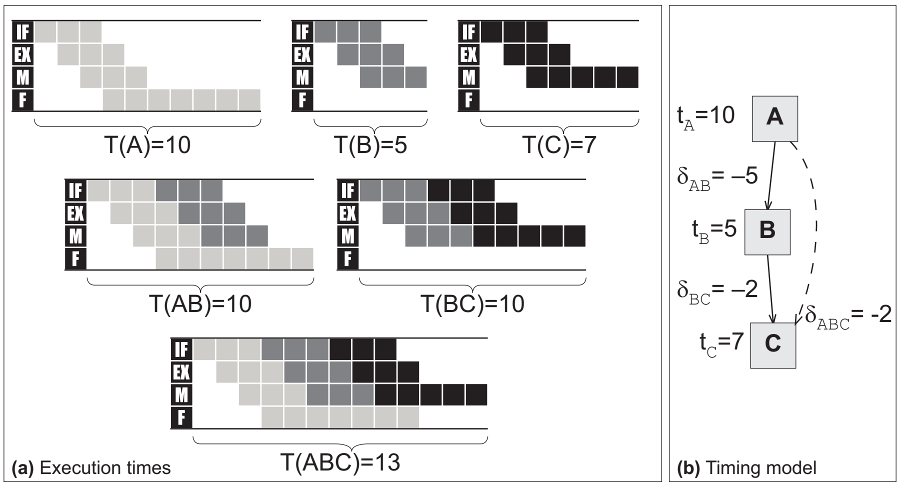


Figure 4.4: Pipeline overlap over three nodes and its timing model

4.3.2 Long Timing Effects

Timing effects over three nodes or more are sometimes needed to accurately model the timing of instructions executing on a pipelined processor. Such effects not only affect the WCET estimate generated, but can also change the construction of the longest path in a program [SEE01a].

Figure 4.4 shows an example where node `B` is completely overlapped by node `A`, and where node `C` will also overlap with node `A`, since there is an instruction in `A` that keeps executing past the end of `B`. The effect on the program timing of this behavior is modeled using a negative timing effect over the sequence `ABC`, as illustrated in Figure 4.4(b).

Figure 4.5 shows an example where a node causes a *delay* to a node that is not an immediate successor. Node `C` uses floating point and gets delayed due to the use of the floating point stage in node `A`, despite the presence of node `B` between them (which does not use floating point). This case is modeled by a positive timing effect for the sequence `ABC`.

Both the above examples are caused by *long-running instructions*, instructions using the pipeline stages in such a way that other instructions can keep

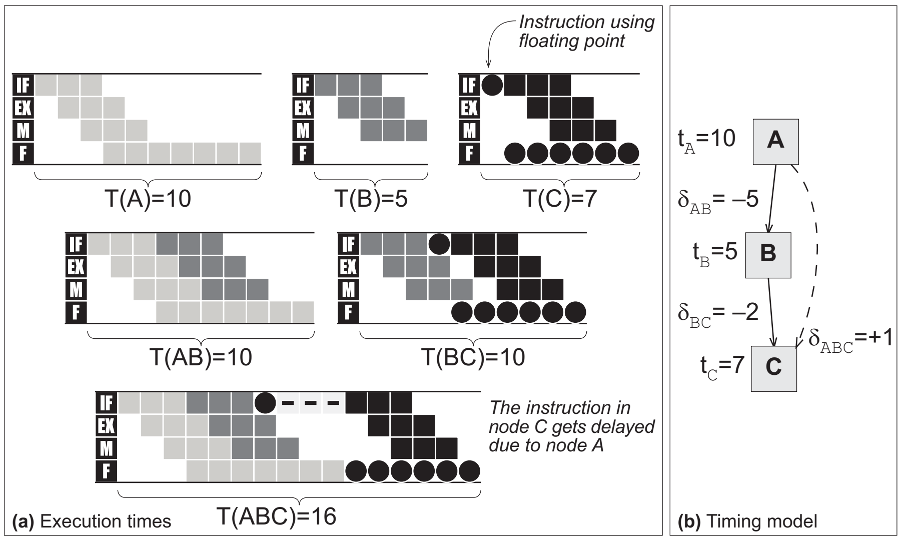


Figure 4.5: Pipeline interference over three nodes and its timing model

executing and completing without being disturbed by the instruction. Thus, a node containing such an instruction can run for a long time in parallel to successive nodes.

For long-running instructions to be present, the completion (but not issue) of instructions must be allowed to be performed out of order. A common case is the use of a floating-point unit, since it usually uses a register bank different from that of integer instructions, and thus there is no data dependence between integer and floating-point instructions. For example, consider the MIPS R4000 [Hei94], where the integer and floating point pipelines execute in parallel[1]. Some floating point instructions require up to 100 cycles for execution, making it possible for long timing effects to appear due to integer instructions completing their execution while the floating point unit is executing the long floating point instruction.

Another case when long timing effects can appear is when a pipeline is structured in such a way that data hazards can appear between non-adjacent instructions. This can be the case if the stage that produces a required value is not adjacent to the stage that requires it (otherwise, the effect would be covered by a pairwise timing effect). One pipeline with this property is shown in Figure 1.7. We assume that all values are written in the *write back* stage `W`, and read in the *register read* stage `RR`. This means that an instruction can be forced to wait for an instruction that comes several steps before it in the instruction stream, as illustrated in Figure 4.6. Note that this type of effect is quite rare, since most modern pipeline designs include "data forwarding"

[1]All instructions pass through the same instruction fetch and decode stages before splitting into the floating point or integer pipeline.

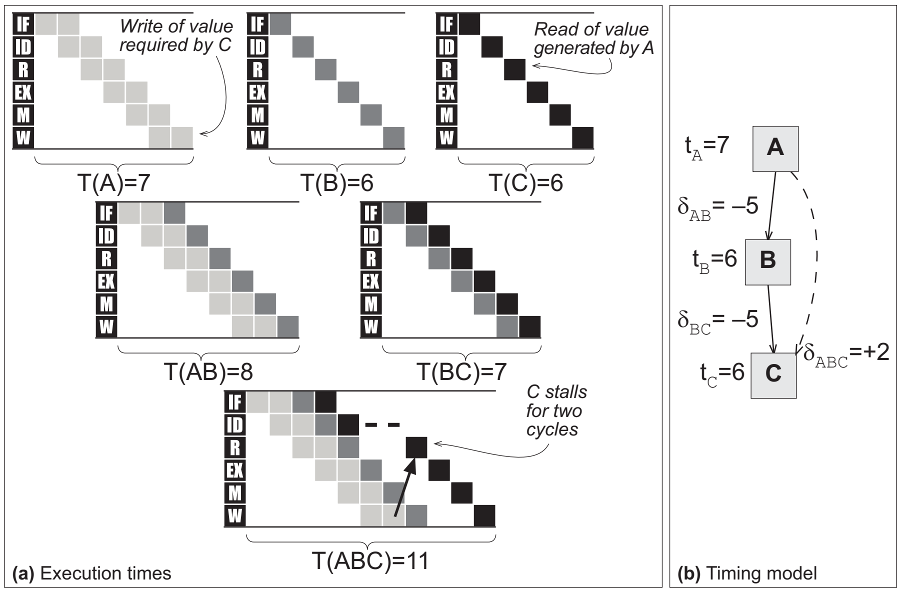


Figure 4.6: Long timing effect due to data hazard

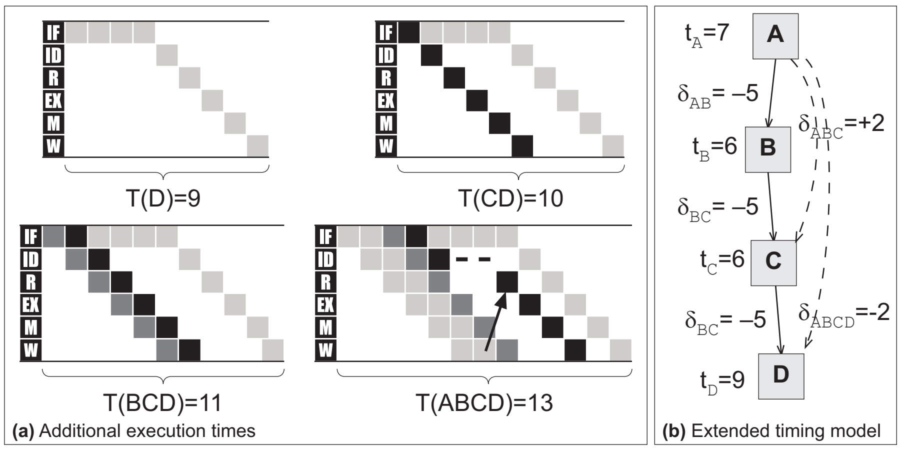


Figure 4.7: Second long timing effect in same sequence

features to reduce the potential stalls due to waiting for data (generated data is made available before the write back stage of the pipeline).

If we extend the example in Figure 4.6, we can get another long timing effect starting with the same initial node `A`. As shown in Figure 4.7, if we add another node `D` to the sequence, we will get a negative timing effect over `ABCD`, where we in effect regain the time lost in the data stall over `ABC`.

Note that even very simple pipelines can exhibit long timing effects. As shown in Figure 4.8, the two nodes `A` and `C` form "complementary" pipeline dia-

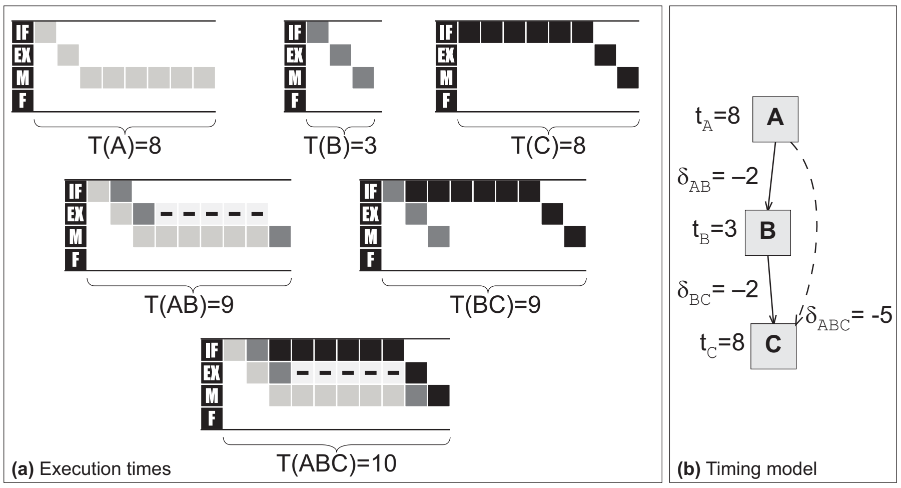


Figure 4.8: Long timing effect in a simple pipeline

grams, and thus can be pushed together very far, if the intermediate node `B` has the right structure. The example is a little contrived, but the long instruction fetch time at the start of node `C` will appear if some areas of memory are slower than others, or if an instruction cache miss occurs.

Chapter 5

Pipeline Model

To show the relationship between the timing model of Chapter 4 and actual hardware, we need to have a formal model for reasoning about the pipeline behavior. This chapter introduces such a model, and proves a number of properties of pipelines.

Note that the illustrations in this chapter use a number of different pipeline organizations. For each illustration, we have strived to find a minimal pipeline that allows us to illustrate a concept, in order to reduce the complexity of the examples, which instead increases the number of different pipeline organizations.

5.1 Modeling Pipelines

We will start by giving a model for a single in-order pipeline, and then show how the model can be generalized to multiple pipelines and more complex pipeline organizations. The modeling technique is not intended to be used as a processor simulator, only to describe the effects of pipelined execution. In particular, for multiple pipelines we assume that it is decided which pipeline a certain instruction uses, and that all data dependences have been discovered.

5.1.1 Single In-Order Pipelines

We use the following model of a processor pipeline: a processor pipeline consists of n *pipeline stages*. Each *instruction* I_i is considered to be a sequence $r_i^1 \dots r_i^n$ of resource requirements, where r_i^j corresponds to the time the execution of the instruction requires in stage j. Only one instruction can occupy a pipeline stage at any particular point in time. Instructions are numbered 1 to m, all instructions will use all pipeline stages, and use them in the same order. Instructions proceed to the next stage as soon as possible. Time is discrete and expressed in clock cycles.

Proposition 5.1 $T(I_i) = \sum_{j=1}^{n} r_i^j$ *for a certain instruction* i*.*

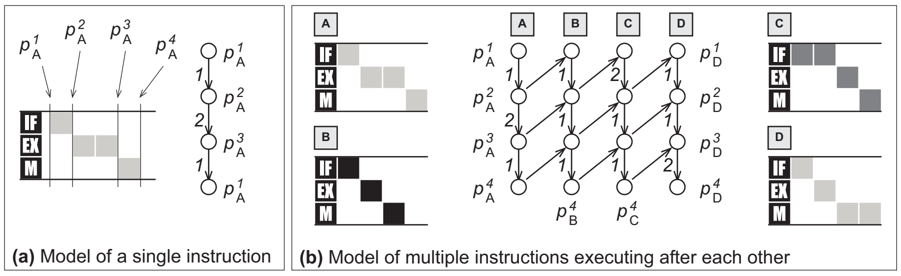

(a) Model of a single instruction
(b) Model of multiple instructions executing after each other
IF
EX
M
A
B
C
D

Figure 5.1: Constraint model of pipeline execution

Proof: Since an instruction has to progress through all stages to complete its execution, the execution time is obviously the sum of the time spent in all the stages. □

To model the timing behavior of a sequence of instructions $I_1 \dots I_m$, we use *constraints* which express conditions on the points in time when instructions enter pipeline stages. For an instruction I_i, let p_i^j be the point in time at which I_i enters the pipeline stage j. By convention, p_i^{n+1} is the time at which the instruction leaves the last stage of the pipeline. For example, Figure 5.1(a) shows the points for the instruction `A` in a three-stage pipeline. $p_{\mathtt{A}}^1$ is the point where `A` enters the pipeline, and $p_{\mathtt{A}}^4$ is the point at which it leaves the pipeline.

The constraints required to model the behavior of a basic pipeline are the following:

$$p_i^{j+1} \geq p_i^j + r_i^j \qquad (1 \leq i \leq m, 1 \leq j \leq n) \qquad (5.1)$$

$$p_{i+1}^j \geq p_i^{j+1} \qquad (1 \leq i < m, 1 \leq j \leq n) \qquad (5.2)$$

Equation (5.1) models the fact that an instruction cannot enter its next stage before the current stage is completed, and Equation (5.2) models the fact that the next instruction cannot enter a certain pipeline stage before the current instruction has started its next stage (thus making sure that only one instruction occupies a certain stage at each point in time). All constraints for a particular sequence of instructions are gathered into a constraint system C.

We can graphically represent this constraint system as a weighted directed acyclic graph where the nodes correspond to the points p_i^j and the arrows correspond to the constraints between the points. Each instruction I_i is drawn as a column of points $p_i^1 \dots p_i^{n+1}$, with the constraints from Equation (5.1) shown as vertical arrows. The weight between p_i^j and p_i^{j+1} corresponds to r_i^j. The constraints from Equation (5.2) are drawn as diagonal arrows with no weight, since they have weight zero. An example is shown in Figure 5.1(b) (we only show some of the p_i^n variables to reduce the clutter).

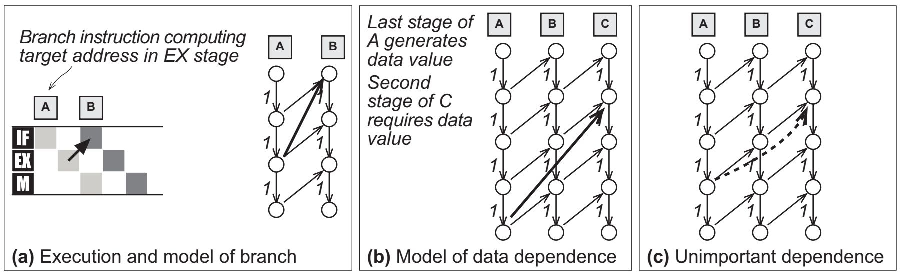

Branch instruction computing target address in EX stage
A
B
IF
EX
M
1
(a) Execution and model of branch
Last stage of A generates data value
Second stage of C requires data value
C
(b) Model of data dependence
(c) Unimportant dependence

Figure 5.2: Constraints for branches and data dependencs

5.1.2 Branch Instructions

Branch instructions potentially break the regular structure of the basic model in Section 5.1.1, since they generate dependences between the end of the stage where the branch is decided (and the target address of the branch computed) and the fetch of the next instrution. Thus, a branch will generate a constraint from some point in the branch instruction to the first point of the next instruction, as illustrated in Figure 5.2(a).

If the branch is computed in stage j of instruction I_i, the constraint generated is the following (since branch dependences always affect the following instruction for an in-order pipeline):

$$p_{i+1}^{1} \geq p_{i}^{j+1} \tag{5.3}$$

5.1.3 Data Dependences

We can also allow for data dependences like those illustrated in Figure 4.6. A data dependence means that before instruction I_i can enter stage j, some previous instruction I_k will have to complete stage l (since stage l computes some data required in stage j). Thus, we get the constraint:

$$p_{i}^{j} \geq p_{k}^{l+1} \tag{5.4}$$

An example of an additional constraint due to a data depedence is shown in Figure 5.2(b), as a long arrow with weight zero.

Note that a data-dependence constraint is only meaningful if it goes between two points that are not otherwise transitively connected via the basic constraints from Equation (5.1) and Equation (5.2). As shown in Figure 5.2(c), such a data dependence will be subsumed by the regular constraints from Equation (5.2).

5.1.4 Multiple In-Order Pipelines

The pipeline model presented in Section 5.1.1 is realistic and corresponds to the pipelines described as simple scalar pipelines in Section 1.7. We can extend the model to the case of scalar and superscalar pipelines, i.e. pipelines that fork out to multiple pipelines at some point. In this extended model, not all instructions

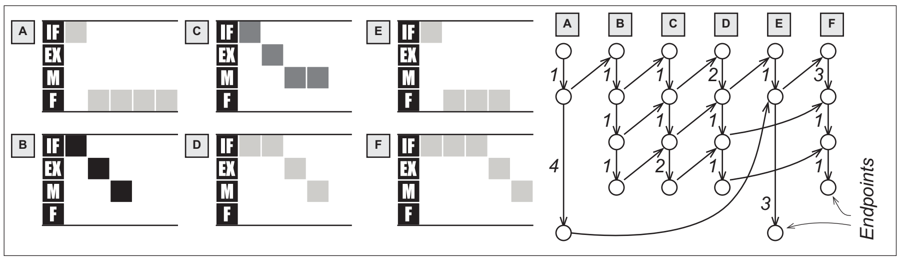


Figure 5.3: Constraints for multiple pipelines

will use all stages, and thus not all instructions will have the same sets of points. More precisely, each instruction will have points corresponding to their entry into the pipeline (p_i^1), and points corresponding to the entry into each stage of the pipeline that they actually use.

The constraint system equations can be reformulated for multiple pipelines by replacing the increments for instruction and pipeline stage numbers. To this end, we define the functions $\texttt{previ}(i, j)$ and $\texttt{nexti}(i, j)$ which report for instruction I_i the previous and next instruction using pipeline stage j. We also have the functions $\texttt{prevs}(i, j)$ and $\texttt{nexts}(i, j)$ that for a certain instruction I_i and pipeline stage j reports the previous and next pipeline stage used by I_i. Thus, for a certain point in the constraint system p_i^j we get:

$$p_i^{\texttt{nexts}(i,j)} \geq p_i^j + r_i^j \tag{5.5}$$

$$p_{\texttt{nexti}(i,\texttt{prevs}(i,j))}^{\texttt{prevs}(i,j)} \geq p_i^j \tag{5.6}$$

Equation (5.5) corresponds to Equation (5.1), that an instruction needs to finish one stage before progressing to the next stage. Equation (5.6) corresponds to Equation (5.2), and forces an instruction to wait until the previous instruction using a stage it needs has left it.

Note that to ensure an acyclic constraint system, we require that the pipeline stages form an acyclic graph (which is always the case in practice), and that instructions are processed in order. The processing in order is expressed as a requirement on the $\texttt{previ}(i, j)$ function:

$$\texttt{previ}(i, j) < i < \texttt{nexti}(i, j)$$

5.1.5 Multiple Pipelines With Joins

Figure 7.1 on page 81 shows that the pipeline of the NEC V850E has two parallel pipelines with the potential for instructions to *cross over* from the master pipeline to the additional pipeline between the `EX` and `MEM` stages. Such flows could potentially cause problems. For example, an instruction executing in the path `IF-ID-EX-MEM` can collide with an instruction executing `IF-ID-MEM` (the lower path), possibly creating cycles in the constraint graph.

However, the V850E avoids such problems by always letting the instruction that entered the pipeline first proceed first. Thus, all constraints still only go from instructions to following instructions, and we have a constraint system which is still an acyclic graph. Note that this is true because the pipeline has such a structure, and instructions execute in such a manner, that an instruction going the shorter path via the lower `ID` will never overtake an instruction taking the longer path via the upper `ID` and `EX` to `MEM`. At worst, the two instructions will reach `MEM` simultaneously.

5.1.6 VLIW Pipelines

As explained in Section 1.7.4, in a VLIW machine, each instruction fetched actually consists of a number of parallel operations, each destined for a single special-purpose pipeline. After instruction fetch is complete, each such pipeline will in generate a constraint system of its own, making a basic VLIW model quite simple. Instruction fetch is handled by making sure that *all* operations in the previous instruction have left the instruction fetch stage before the next instruction can enter.

In many cases, the pipelines are not completely decoupled. If hardware interlocks to protect data dependences are not used in order to simplify implementation [WB98, TI00], the VLIW instructions have to proceed in lock-step in the pipeline: no operation in a VLIW instruction can be allowed to race ahead of the other operations in the same instruction. An example is the total pipeline stop caused by data load latencies on the Texas Instruments TMS320C64x [TI00, Section 6.3.3.1]. This will generate a series of constraints in that an operation in instruction I_{i+1} cannot enter its next stage until all operations in I_i have left their current stage. Note that it is easy to make a system more regular by adding extra points in the constraint system that corresponds to the point in time when an entire very-long instruction enters its next pipeline stage. This also gives us one unique starting point for the constraint system.

5.1.7 Superscalar Pipelines

A concrete execution of a sequence of instructions in a superscalar pipeline can be modeled using our constraint model. The common part of the pipeline (typically, fetch and dispatch stages), where all instructions proceed in parallel, can be modeled as single stages, with a high fanout from the last common stage to the start of the execution pipelines.

However, most of the properties proven in Section 5.2 rely on the fact that adding a single instruction I_1 in front of a sequence of instructions $I_2 \ldots I_m$ does not change the structure of the constraint system for $I_2 \ldots I_m$. But the dynamic grouping of instructions in superscalars will probably make the constraints for $I_2 \ldots I_m$ on their own different from the constraints for $I_2 \ldots I_m$ that is part of

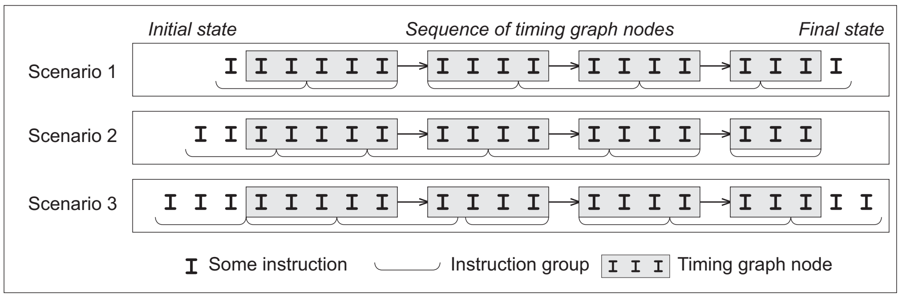


Figure 5.4: Example of potentially infinite state propagation in superscalar

$I_1 \dots I_m$, since the instruction grouping starting with I_1 can be different from that starting with I_2.

The superscalar instruction grouping is to some extent orthogonal to the basic pipeline behavior, and it can cause potentially infinite propagation of state in the pipeline. This is illustrated in Figure 5.4: assuming a processor that can issue three instructions per clock cycle, the scenarios presented show that over a series of nodes where the scheduler manages to issue three instructions per cycle, the initial state of the scheduler affects the final state. In “scenario 1”, the first two instructions in the sequence are grouped with a single preceding instruction, and at the end of the sequence there is one free slot in the final group. In “scenario 2”, the first instruction is grouped with two preceding instructions, and the last instruction group is completely filled, while “scenario 3” illustrates that yet another final state is obtained if the first node starts a new instruction group.

5.2 Properties of Pipelines

This section uses the pipeline model presented above to prove a number of properties about pipelines that are of relevance to worst-case execution time analysis. We start by defining some useful concepts, and then go on to prove various properties.

Note that the results shown here are only applicable if the hardware model used measures time like we do in the pipeline model, that is, from the time the first instruction enters the pipeline to the time the last instruction exits the pipeline.

5.2.1 Useful Concepts

A *perfect instruction* I_i is an instruction that spends only one cycle in each stage (all $r_i^j = 1$).

A *long constraint* is a constraint from a point in instruction I_i to a point in instruction I_j, where $j > i + 1$, that is, a constraint between two non-

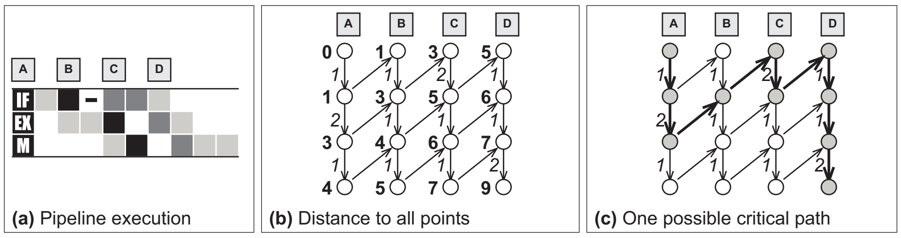


Figure 5.5: Pipeline execution, constraint solution, and critical path for the example in Figure 5.1

adjacent instructions. Data dependences and parallel pipelines can cause long constraints.

The *endpoints* of a constraint graph are the points with no outgoing arrows. Figure 5.3 shows an example constraint system for a multiple-pipeline processor with the endpoints marked.

Final parallelism occurs between an instruction I_1 and a sequence of following instructions $I_2 \dots I_m$ when there is no constraint from the last point of I_1 to any instruction $I_2 \dots I_m$, i.e., the last point of I_1 is an endpoint. An example is given in Figure 5.3, where instruction `A` is finally parallel to `BCD` (but not to the full sequence `BCDEF`, since `E` also uses the floating point unit).

A *path* P in the constraint system is a sequence of points $p_{i_1}^{j_1}, p_{i_2}^{j_2}, p_{i_3}^{j_3} \dots$ such that for any two successive points $p_{i_k}^{j_k}, p_{i_{k+1}}^{j_{k+1}}$ in P, there is a constraint (arrow) from $p_{i_k}^{j_k}$ to $p_{i_{k+1}}^{j_{k+1}}$.

The *length*(P) of a path P is the sum of the weights of all arrows in P. Since only arrows corresponding to Equation (5.1) have weight, it is the sum of all r_i^j values along the path.

We define *paths*(p_i^j, p_k^l) to be the set of all paths from the point p_i^j to the point p_k^l.

A point p_k^l in the constraint graph can be *reached* from point p_i^j if there is a path in the constraint graph from p_i^j to p_k^l.

The *distance* between two points p_i^j and p_k^l in the constraint graph is denoted by $D(p_i^j, p_k^l)$. The distance is defined as the maximal length of all possible paths from p_i^j to p_k^l (note that the distance can only be constructed if p_k^l can be reached from p_i^j):

$$D(p_i^j, p_k^l) = \max_{P \in paths(p_i^j, p_k^l)} (length(P)) \tag{5.7}$$

A *critical path* CP is a path between two points p_i^j and p_k^l such that $D(p_i^j, p_k^l) = length(CP)$, that is, a path of maximal length. An example of a critical path is shown in Figure 5.5(c). Note that there may exist several critical paths between two points in a constraint system.

Proposition 5.2 *The execution time $T(I_1 \dots I_m)$ of a sequence of instructions $I_1 \dots I_m$ forming a constraint system C is the maximal distance from p_1^1 to some point in C:*

$$T(I_1 \dots I_m) = max(D(p_1^1, p_i^j)), \quad (1 \leq i \leq m, 1 \leq j \leq n)$$

Proof: We will start by proving that for each point p_i^j, the distance $D(p_1^1, p_i^j)$ corresponds to the time at which instruction I_i enters stage j (if we assume that the first instruction enters the pipeline at time zero). We perform the proof by induction over the maximal number of arrows in any path from p_1^1 to p_i^j.

Base case: For the point p_1^1, the statement is trivially true.

Inductive case: Consider a point p_i^j. It has arrows from a number of points p_k^l, at which, by the induction hypothesis, $D(p_1^1, p_k^l)$ is the time when instruction I_k enters stage l of the pipeline. Since all arrows correspond to $\geq$-constraints, $D(p_1^1, p_i^j)$ will be the maximum of the incoming arrows (including their weight).

Considering the execution of instructions in a real processor pipeline, an instruction I_i will enter stage j at the first point in time that the following constraints are true: I_i has completed its own previous stage, stage j has been cleared by the previous occupant, and all data required has been delivered.

Since the distance is formed by taking the largest value of the preceding points (plus the weight of arrow, where applicable), $D(p_1^1, p_i^j)$ corresponds to the time that instruction I_i enters stage j of the pipeline, and the inductive case is proven.

Hence, the distance $D(p_1^1, p_i^{last})$ to the last point p_i^{last} of an instruction I_i corresponds to the time that I_i leaves the pipeline, and the point p_i^{last} with the highest value $D(p_1^1, p_i^{last})$ corresponds to the point in time when all instructions have left the pipeline. $\square$

For single in-order pipelines, there is only one endpoint for a sequence of instructions $I_1 \dots I_m$, p_m^{n+1}, and the execution time is $D(p_1^1, p_m^{n+1})$. The relation between execution time in the pipeline and the distances in the constraint system is illustrated in Figure 5.5(a) and (b).

It is easy to see that Proposition 5.1 follows from Proposition 5.2 in the special case that $m = 1$. There is only one path from p_i^1 to p_i^{n+1}, and the distance $D(p_i^1, p_i^{n+1})$ is obviously the sum of the resource times r_i^j for the instruction.

We also note that the constraint system can be considered a scheduling problem in an acyclic graph, and in scheduling theory it is well-known that an ASAP (as-soon-as-possible) schedule always gives an the optimal solution for such problems. It also known that for such graphs, the total execution time is the length of the critical path, which fits nicely with our concepts and the proof of Proposition 5.2.

A *pipeline stall* can be defined as shown in Equation (5.8):

$$D(p_1^1, p_i^j) > D(p_1^1, p_i^{\texttt{prevs}(i,j)}) + r_i^{\texttt{prevs}(i,j)} \tag{5.8}$$

That is, whenever the distance $D(p_1^1, p_i^j)$ is greater than the time obtained from the previous stage of the same instruction. The instruction would like to continue with its next pipeline stage, but is prevented from doing so since it has to wait for an earlier instruction to vacate the next pipeline stage or to complete a data dependence. This covers both data hazards and structural hazards.

An instruction I_i *stalls another instruction* I_k if there is a stall at a point p_k^l in I_k, and there is an arrow from a point p_i^j in I_i such that it is the time value propagated along the arrow gives p_k^l its value (the greatest of all predecessor times). In pipeline terms, the instruction I_i blocks the instruction I_k from entering its next stage due to a structural or data hazard.

5.2.2 Effects Between Neighbors

Using this model of pipelined execution, only negative timing effects can occur between two neighboring instructions I_1 and I_2, as shown in Theorem 5.1.

Theorem 5.1 *For any two successive instructions I_1 and I_2, the timing effect $\delta_{I_1 I_2} \leq 0$.*

Proof: In the worst case, the critical path in the constraint system C for I_1 and I_2 will include all r_i^j values, and thus $T(I_1 I_2) \leq \sum_{j=1}^{n} r_1^j + \sum_{j=1}^{n} r_2^j$. By Proposition 5.1, $T(I_1) = \sum_{j=1}^{n} r_1^j$ and $T(I_2) = \sum_{j=1}^{n} r_2^j$. Thus, $T(I_1 I_2) \leq T(I_1) + T(I_2)$, and by Equation (4.2), $\delta_{I_1 I_2} \leq 0$. $\square$

5.2.3 Pipelines Without Long Timing Effects

There are some classes of single pipelines where no long timing effects can occur at all. Theorem 5.2 shows the result for an unrealistic perfect pipeline where each instruction takes a single cycle in each stage, and no data dependences occur. Unfortunately, such pipelines are rather rare in real life.

By allowing a single pipeline stage to take more time, we get a pipeline like that of the ARM7 [ARM95], where instructions progress through a few single-cycle pipeline stages for instruction fetch and decoding before entering an execute stage where several cycles can be spent. Theorem 5.3 shows that no long timing effects can occur for such a pipeline. In the common case that the execution stage is the stage with variable duration (as on the ARM7), Theorem 5.3 demands that single-cycle instruction memory is used and that all data dependences are between the variable-length stages of adjacent instructions. Otherwise, long timing effects like those shown in Figure 4.6 and Figure 4.8 could conceivably occur.

Theorem 5.2 *For a single pipeline with n stages where every instruction is perfect (every instruction uses each pipeline stage for precisely a single cycle), with no data dependences or branch dependences, no long timing effects can occur.*

Proof: For such a pipeline with n stages, the time to execute m instructions is $n+m-1$, since the first instruction takes n cycles and each following instruction adds a single clock cycle of execution time. In this case, we get the following (assuming a single instruction in each node):

$$\begin{aligned}
T(I_1 \ldots I_m) &= n+m-1 \\
t_{I_i} &= n \\
\delta_{I_1 I_2} &= T(N_1 N_2) - T(N_1) - T(N_2) + T() \\
&= n+2-1-(n+1-1)-(n+1-1)+0 = -(n-1) \\
\delta_{I_1 \ldots I_m, m \geq 3} &= T(I_1 \ldots I_m) - T(I_1 \ldots I_{m-1}) - T(I_2 \ldots I_m) + T(I_2 \ldots I_{m-1}) \\
&= n+m-1-(n+m-2)-(n+m-2)+n+m-3 = 0
\end{aligned}$$

Thus, no long timing effects can occur. $\square$

Theorem 5.3 *For a single pipeline with n stages, all of which take one cycle except for stage v that takes one or more cycles, no long timing effects can occur.*

Proof: Assume that the execution time of an instruction I_i in the variable-length stage is r_i^v. Then, the execution time of a single instruction I_i is $n-1+r_i^v$. The execution time for a sequence of instructions $I_1 \ldots I_m$ is $n-1+r_1^v+\cdots+r_m^v$. We get the following calculations:

$$\begin{aligned}
T(I_1 \ldots I_m) &= n-1+r_1^v+\cdots+r_m^v \\
t_{I_1} &= n-1+r_1^v \\
\delta_{I_1 I_2} &= T(I_1 I_2) - T(I_1) - T(I_2) + T() \\
&= n-1+r_1^v+r_2^v-(n-1+r_1^v)-(n-1+r_2^v)+0 = -(n-1) \\
\delta_{I_1 \ldots I_m, m \geq 3} &= T(I_1 \ldots I_m) - T(I_1 \ldots I_{m-1}) - T(I_2 \ldots I_m) + T(I_2 \ldots I_{m-1}) \\
&= n-1+r_1^v+\ldots+r_m^v-(n-1+r_1^v+\ldots+r_{m-1}^v) \\
&\quad -(n-1+r_2^v+\ldots+r_m^v)+(n-1+r_2^v+\ldots+r_{m-1}^v) = 0
\end{aligned}$$

Thus, no long timing effects can occur. $\square$

If we look at the critical paths for the case in Theorem 5.3, we note that a critical path will always follow the v stage.

5.2.4 Pipelines Without Positive Long Timing Effects

For single in-order pipelines with branches and data dependences between adjacent instructions, no positive long timing effects can appear. This result means that we get a safe overestimation of the execution time of a program by only considering timing effects for sequences shorter than a certain length l. By increasing l, we only get better estimates. As shown in Section 5.2.6, in general

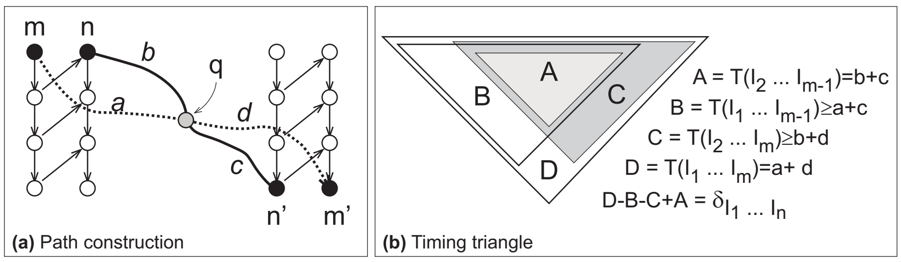


Figure 5.6: Illustrating the principles of Theorem 5.4

we cannot assume that there is some upper bound on the maximal length of sequences that can cause negative timing effects.

The proof is performed in two stages. Theorem 5.4 shows a basic result for a pipeline model built from Equation (5.1) and Equation (5.2). Theorem 5.5 shows the result for the more complex case also including branch dependences (from Equation (5.3)), and data dependences between adjacent instructions (from Equation (5.4)).

Note that the applicability of this result is limited to the case of single pipelines with data depedences only between adjacent instructions. If we allow data dependences between non-adjacent instructions, we can get positive timing effects as shown in Figure 4.6. If we allow multiple pipelines, positive timing effects as shown in Figure 4.5 can occur.

Theorem 5.4 *For a sequence of instructions $I_1 \ldots I_m, m \geq 2$, executing on a single in-order pipeline without branch dependences or data dependences, $\delta_{I_1 \ldots I_m} \leq 0$.*

Proof: Consider the relation between the execution times and the constraint system formed by $I_1 \ldots I_m$. As shown in Figure 5.6(a), we have four points:

- m, corresponding to the start of instruction I_1
- n, corresponding to the start of I_2
- n', corresponding to the end of I_{m-1}
- m', corresponding to the end of I_m

The distances between these points form the execution times involved in the $\delta_{I_1 \ldots I_m}$ calculation as follows:

$$\begin{aligned}
T(I_1 \ldots I_m) &= D(m, m') = \mathit{length}(CP(m, m')) \\
T(I_2 \ldots I_{m-1}) &= D(n, n') = \mathit{length}(CP(n, n')) \\
T(I_1 \ldots I_{m-1}) &= D(m, n') = \mathit{length}(CP(m, n')) \\
T(I_2 \ldots I_m) &= D(n', m) = \mathit{length}(CP(n, m')) \\
\delta_{I_1 \ldots I_m} &= D(m, m') + D(n, n') - D(m, n') - D(n, m')
\end{aligned}$$

We must show that $D(m, m') + D(n, n') \leq D(m, n') + D(n, m')$. We select one of the critical paths from n to n' ($CP(n, n')$), which is shown as the solid-drawn

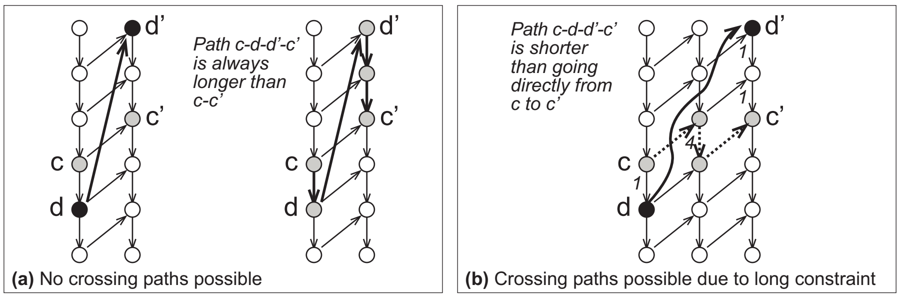

d'
Path c-d-d'-c' is always longer than c-c'
c'
c
d
d'
c'
c
d
(a) No crossing paths possible
Path c-d-d'-c' is shorter than going directly from c to c'
d'
1
1
c'
4
c
1
d
(b) Crossing paths possible due to long constraint

Figure 5.7: Illustrating the principles of Theorem 5.5

line in Figure 5.6(a). We also select one of the critical paths from m to m' ($CP(m, m')$), shown as a dashed line Figure 5.6(a).

The crucial fact is that $CP(n, n')$ and $CP(m, m')$ cross in a point q (the location of this point depends on the structure of the constraint system). This point divides $CP(n, n')$ into two paths segments, $P(n, q)$ with length $b = D(n, q)$ and $P(q, n')$ with length $c = D(q, n')$. Likewise, $CP(m, m')$ is divided into two segments with lengths $a = D(m, q)$ and $d = D(q, m')$.

We now need to determine $D(m, n')$ and $D(n, m')$. Using the two path segments $P(m, q)$ and $P(q, n')$, we construct a path from m to n'. Since we can get from m to n' in this manner, any critical path $CP(m, n')$ must be as long or longer than this path. This gives that $D(m, n') \geq a + c$. The same reasoning can be applied to paths from n to m', obtaining $D(n, m') \geq b + d$

Thus, we get the following distances and thereby execution times:

$$\begin{aligned} D(m, m') &= a + d \\ D(n, n') &= b + c \\ D(m, n') &\geq a + c \\ D(n, m') &\geq b + d \end{aligned}$$

Since $D(m, n') + D(n, m') \geq D(m, m') + D(n, n')$, we get $\delta_{I_1 \ldots I_m} \leq 0$. □

Theorem 5.5 *For a sequence of instructions $I_1 \ldots I_m, m \geq 2$, executing on a single in-order pipeline with branch dependences and data dependences between adjacent instructions, $\delta_{I_1 \ldots I_m} \leq 0$.*

Proof: To prove this theorem, we rely on the construction in Theorem 5.4, and show that it can still be achieved in this case.

For the construction to be invalidated, we must be able to find two critical paths $CP(n, n')$ and $CP(m, m')$ such that no common point q can be found. For this to occur, the paths must cross without sharing a common point in the constraint system. This can only happen at the new arrows introduced by the data dependences between adjacent instructions and the branch dependences, since they cross over the basic arrows of the constraint system (as illustrated by the arrow from d to d' in Figure 5.7).

These crossing arrows do not allow critical paths to cross without sharing a node. Consider the case in Figure 5.7(a), where the arrow from c to c' crosses the arrow from d to d'. However, the arrow from c to c' cannot be on a longest path (critical path), since if it was, that path could be made longer simply by going from c to d to d' to c', and the critical paths would then share a number of nodes.

Hence, we cannot construct $CP(n, n')$ and $CP(m, m')$ in Figure 5.6 such that no common point q is present. Thus Theorem 5.4 holds also in the presence of branches and data dependences between adjacent instructions. □

Note that if we have data dependences across more than two instructions, the case in Figure 5.7(b) can occur, and the two paths can cross without sharing a node.

5.2.5 Source of Long Timing Effects

A long timing effect for a sequence of instructions $I_1 \ldots I_m, m \geq 3$, occurs whenever I_1 has the effect of disturbing the execution in such a way that the execution of the instructions $I_2 \ldots I_m$ is different compared to if I_1 had not been present. This occurs precisely when I_1 causes a *stall* to some instruction following it, which is proven in Theorem 5.6. The theorem gives a necessary condition for the appearance of timing effects, but not a sufficient condition. It is possible that no timing effect materializes at all, even though a stall is present (as illustrated in Figure 5.13). Theorem 5.6 does not indicate whether the effect that could appear is positive or negative.

Proposition 5.3 *If an instruction I_1 does not stall any succeeding instructions $I_2 \ldots I_m$, then $D(p_1^1, p_i^j) = D(p_2^1, p_i^j) + r_1^1$.*

Proof: Since there is no stall from I_1, the maximum time for a point in $I_2 \ldots I_m$ is never going to come across an arrow from I_1, except for the case of p_2^1, since the only incoming arrow is the one from p_1^2 (it has to wait for instruction I_1 to clear the first stage of the pipeline before entering). Thus, all longest paths from p_1^1 can be constructed by first going to p_1^2, and then to p_2^1. The extra time required to go from p_1^1 is then just r_1^1. □

Theorem 5.6 *A timing effect $\delta_{I_1 \ldots I_m} \neq 0$ can occur for a sequence of instructions $I_1 \ldots I_m, m \geq 3$ only if I_1 stalls the execution of some instruction in $I_2 \ldots I_m$, or I_1 is finally parallel to $I_2 \ldots I_m$.*

Proof: In the case that I_1 is finally parallel to $I_2 \ldots I_m$, a timing effect can occur in the fashion illustrated in Figure 4.4.

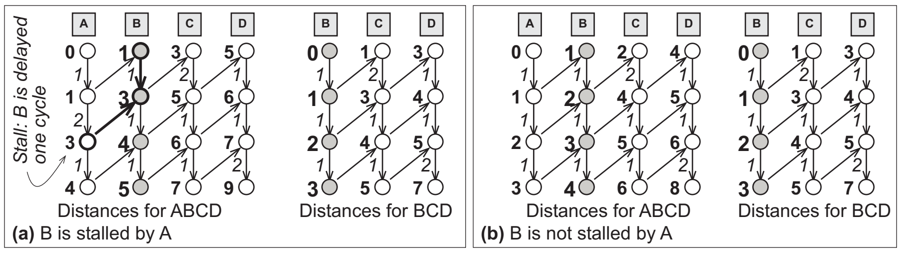


Figure 5.8: Example of difference between stalled and non-stalled execution

In the case that the instruction is not finally parallel, we use Proposition 4.1, which gives us that $\delta_{I_1 \ldots I_m} = 0$ whenever $\epsilon_{N_1 \ldots N_{n-1}|N_n} = \epsilon_{N_2 \ldots N_{n-1}|N_n}$. By Equation (4.4), we get the following:

$$\begin{aligned}\epsilon_{I_1 \ldots I_{m-1}|I_m} - \epsilon_{I_2 \ldots I_{m-1}|I_m} = \;& T(I_1 \ldots I_m) - T(I_1 \ldots I_{m-1}) \\ & - T(I_2 \ldots I_m) + T(I_2 \ldots I_{m-1})\end{aligned}$$

By Proposition 5.2, we know that the execution time for each sequence is the maximum of the distance from the start point of the sequence to the endpoints in the sequence. Since the case where I_1 is finally parallel has been handled separately, we know that there are no endpoints in I_1. From Proposition 5.3, we can deduce that the same point (or set of points with equal values) will be maximal both when counting an execution time from I_1 and I_2, since the distances $D(p_2^1, p_i^j)$ are a constant offset from the distances $D(p_1^1, p_i^j)$. All together, we get the following:

$$\begin{aligned} T(I_1 \ldots I_m) &= max(q \in I_2 \ldots I_m : D(p_1^1, q)) = a \\ T(I_1 \ldots I_{m-1}) &= max(q \in I_2 \ldots I_{m-1} : D(p_1^1, q)) = b \\ T(I_2 \ldots I_m) &= max(q \in I_2 \ldots I_m : D(p_2^1, q)) = a + r_1^1 \\ T(I_1 \ldots I_{m-1}) &= max(q \in I_2 \ldots I_{m-1} : D(p_2^1, q)) = b + r_1^1 \\ \epsilon_{I_1 \ldots I_{m-1}|I_m} - \epsilon_{I_2 \ldots I_{m-1}|I_m} &= a - b - a - r_1^1 + b + r_1^1 = 0 \end{aligned}$$

$\square$

Thus we have shown that long timing effects only occur if the first instruction of a sequence stalls some successor instruction. An illustration of the idea is shown in Figure 5.8, where Figure 5.8(a) shows a case with a stall from `A` to `B`, and Figure 5.8(b) the same case without a disturbance from `A`.

5.2.6 Upper Bound on Timing Effects

Considering the safety and efficiency of the analysis, it would be very nice if we could provide a bound on the longest sequence of instructions that can generate a timing effect. Unfortunately, no such bound exists in the general case. As shown in Figure 5.9 and Figure 5.10, both negative and positive long timing effects can propagate for unbounded distances.

The cases shown have been carefully constructed, and should be very hard to encounter in real life. But we still need to handle the possibility of unbounded long timing effects in some manner, if we want to create a truly safe analysis.

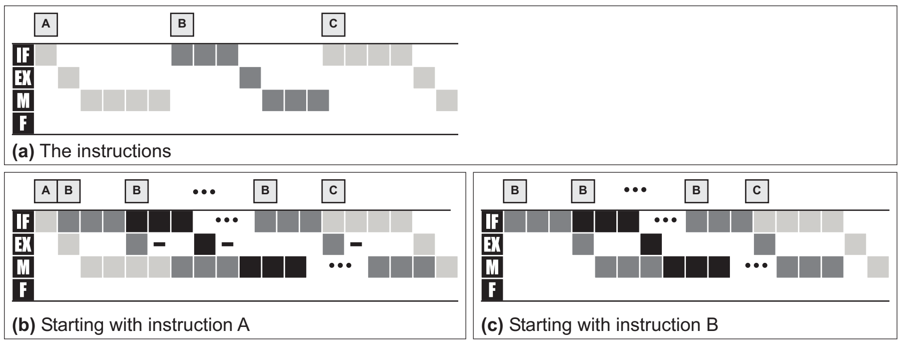

A
B
C
IF
EX
M
F
(a) The instructions
(b) Starting with instruction A
(c) Starting with instruction B

Figure 5.9: Example of an unbounded negative timing effect

Figure 5.9 shows a case where any number of instructions like `B` can be inserted between the instructions `A` and `C`, and we get a timing effect when `C` is executed. To see this, consider the execution times of the sequences, with n denoting the number of times instruction `B` is inserted between `A` and `C`:

$$
\begin{aligned}
T(\mathtt{AB}\ldots\mathtt{BC}) &= 7 + 3n \\
T(\mathtt{AB}\ldots\mathtt{B}) &= 6 + 3n \\
T(\mathtt{B}\ldots\mathtt{BC}) &= 6 + 3n \\
T(\mathtt{B}\ldots\mathtt{B}) &= 4 + 3n \\
\delta_{\mathtt{A}\ldots\mathtt{C}} &= 7 + 3n - 6 + 3n - 6 + 3n + 4 + 3n = -1
\end{aligned}
$$

Thus, we can have a timing effect over an arbitrary distance in the timing graph. The effect results from the fact that the pipeline stall caused by `A` to the first of the `B` instructions when waiting to enter stage `M` propagates via arbitrarily long sequences of `B` instructions, and that no such effect is present when only instructions like `B` are executed. `C` then causes this disturbance to surface as a timing effect. Note that the timing effects for the subsequences `AB`...`B` are always zero.

Positive Long Timing Effects Really spoiling the party, the example in Figure 5.10 demonstrates that due to data dependences, we can get a *positive* timing effect after an arbitrary sequence of instructions. There is a data dependence between instructions `A` and `C` that causes a long constraint to be present. After an arbitary number of instrutions `D`, we get a positive timing effect when instruction `E` is added. From Figure 5.10(a) and Figure 5.10(b) we get the following execution times for the sequences:

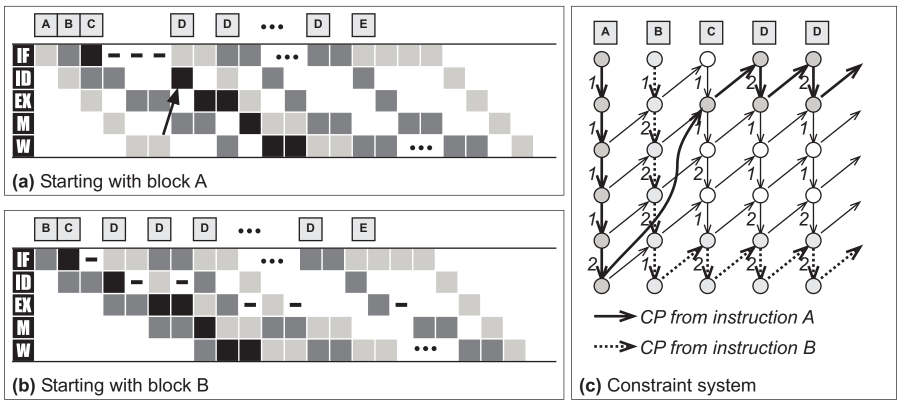


Figure 5.10: Example of an unbounded positive timing effect

$$
\begin{aligned}
T(\texttt{ABCD}\ldots\texttt{DE}) &= 14 + 2n \\
T(\texttt{ABCD}\ldots\texttt{D}) &= 12 + 2n \\
T(\texttt{BCD}\ldots\texttt{DE}) &= 11 + 2n \\
T(\texttt{BCD}\ldots\texttt{D}) &= 10 + 2n \\
\delta_{\texttt{A}\ldots\texttt{C}} &= 14 + 2n - 12 - 2n - 11 - 2n + 10 + 2n = +1
\end{aligned}
$$

Note that in this example, we also have $\delta_{ABC} = -1$ and $\delta_{ABCD\ldots D} = 0$. Thus, the positive timing effect occurs after a number of zero timing effects.

Looking at the constraint graph for the start of the example, as illustrated in Figure 5.10(c), we see the cause of this unbounded timing effect. The data dependence allows for the appearance of *two* critical paths in the constraint graph with equal length. When instruction `E` is added at the end however, the top critical path becomes the only critical path, and the timing effect of the data dependence materializes.

Compared to the construction in Theorem 5.4, we note that the paths from the beginning of instruction `A` and instruction `B` never intersect, and thus there is no point q. This is shown in Figure 5.10(c).

5.2.7 Absence of Timing Anomalies

Lundqvist and Stenström [LS99b] introduced the concept of a *timing anomaly* in WCET analysis. Given a sequence of instructions $I_1 \ldots I_m$ with execution time $a = T(I_1 \ldots I_m)$, if we change the execution time of the first instruction, we get a new execution time $b = T(I'_1 \ldots I_m)$ (with I'_1 different from I'_1). The change in overall execution time is $d = b - a$. If the change to I_1 was an *increase* with k cycles, we have an anomaly if $d < 0$ or $d > k$. If the change to I_1 was a *decrease* by k cycles, we have an anomaly if $d > 0$ or $d < -k$. Note that this property was stated without considering individual pipeline stages, and we translate it to our more detailed model by assuming that precisely one pipeline stage takes longer time for I'_1 compared to I_1.

By example, Lundqvist and Stenström demonstrated that out-of-order processors can suffer from timing anomalies. Here, we show that for pipelines that can be modeled by our constraint systems, such effects cannot occur.

Theorem 5.7 *For in-order pipelines, no timing anomalies can appear when we increase the execution time of an instruction.*

Proof: The original execution time for $I_1 \dots I_m$ corresponds to the longest path in a constraint system C_1. We increase the execution time by increasing the time for I_1 to complete one of its stages, obtaining a new constraint system C_2, where some r_1^j is bigger than in C_1.

The new execution time for $I_1 \dots I_m$ corresponds to some critical path in constraint system C_2. If the arrow with r_1^j was on a critical path before, the execution time will increase by k cycles, $d = k$. If it was not on the critical path before, and now is included, the time will increase by at most k. If it is not on the critical path of either C_1 and C_2, $d = 0$. Thus, $0 \leq d \leq k$, and no timing anomaly can appear. □

Theorem 5.8 *For in-order pipelines, no timing anomalies can appear when we decrease the execution time of an instruction.*

Proof: We assume the same setup as for the proof of Theorem 5.7, except that r_1^j is now smaller in C_2 than in C_1. If the arrow with r_1^j is not on a critical path in C_1, it cannot be so in C_2 either, and the execution time does not change. If the arrow is on the critical path in C_1 and still is in C_2, the execution time has decreased by k cycles. If the arrow is on a critical path in C_1 and not on a critical path in C_2, the execution time has decreased by at most k cycles. Thus, $-k \leq d \leq 0$, and no timing anomaly appears. □

5.2.8 Stabilizing the Pipeline State

As shown in Section 5.2.6, feeding an arbitrary number of instructions through a pipeline does not guarantee that all timing effects from an instruction I_1 have been observed. However, Theorem 5.9 demonstrates that by sending a number of perfect instructions through a pipeline, it is possible to construct a form of barrier, across which no long timing effects can occur.

A consequence of Theorem 5.9 is that if we want to measure the execution time of a sequence of code, after executing some unknown sequence of code, inserting a number of `NOP` instructions allows us to reach a perfect pipeline state, and thus we can measure the execution time without worrying about the initial state causing unexpected delays (this method was used in the validation of our V850E model, see Section 8.2.1).

Theorem 5.9 *For a single in-order pipeline with n stages and allowing data dependences, no timing effects can appear past a sequence of n perfect instructions that have no incoming data dependences (for example, a sequence of* `NOP` *instructions).*

Proof: Due to stalls and data dependences in preceding instructions, the points $p_i^1 \dots p_i^{n+1}$ of the last instruction I_i before the sequence of perfect instructions $I_{i+1} \dots I_{i+n}$ will have different values $a_1 \dots a_{n+1}$. Due to the constraints Equation (5.1), $a_{n+1} \geq a_n \geq \dots \geq a_1$.

In the worst case (which is also the expected case, unless some $r_i^j = 0$), $a_{n+1} > a_n > \dots > a_1$. Now, $p_{i+1}^n = a_{n+1}$, as this is the biggest value it can obtain considering Equation (5.1) and Equation (5.2) (since $r_{p_i}^j = 1$). For I_{i+k}, $p_{i+k}^{n+1-i} = a_{n+1}$, and thus after at most n perfect instructions, $p_{i+n}^1 = a_{n+1}$.

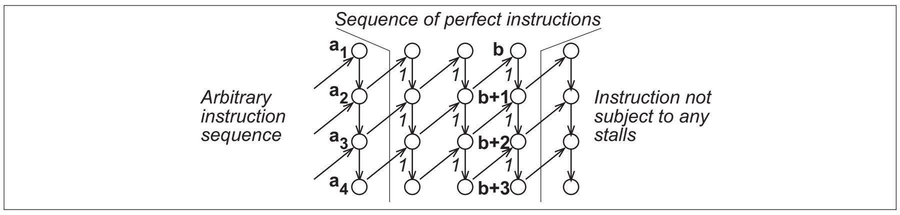

Sequence of perfect instructions
Arbitrary instruction sequence
a1
a2
a3
a4
1
b
b+1
b+2
b+3
Instruction not subject to any stalls

Figure 5.11: Example of cleaning the pipeline with perfect instructions

From p_{i+n}^1, with $D(p_1^1, p_{i+n}^1 = b$, when distances are calculated we get values $b, b+1, \dots, b+n$. This provides a perfect boundary that can cause no stalls to the next instruction, and by Theorem 5.6, there can then be no timing effects from I_{i+n} onwards. The principle is illustrated in Figure 5.11. □

5.2.9 Properties that are not True

During the course of our investigation of pipeline properties, some properties that would have been very useful and helpful for constructing safe WCET analysis methods were formulated and shown to be false. We present the most important false properties here, so that others don't have to repeat the same mistakes.

One Positive Timing Effect Per Disturbance As shown in Figure 5.12, a single stall from instruction `A` to instruction `C` (the stall is caused by instruction `C` waiting to enter pipeline stage `F`, which is occupied by instruction `A`) causes positive long timing effects for both the sequences `ABC` and `ABCDDE`. This example indicates that a single disturbance can give rise to more than one positive long timing effect. In the example, we assume a pipeline with a main path `IF-ID-EX-M-W`, and a path for floating point instructions of `IF-F`.

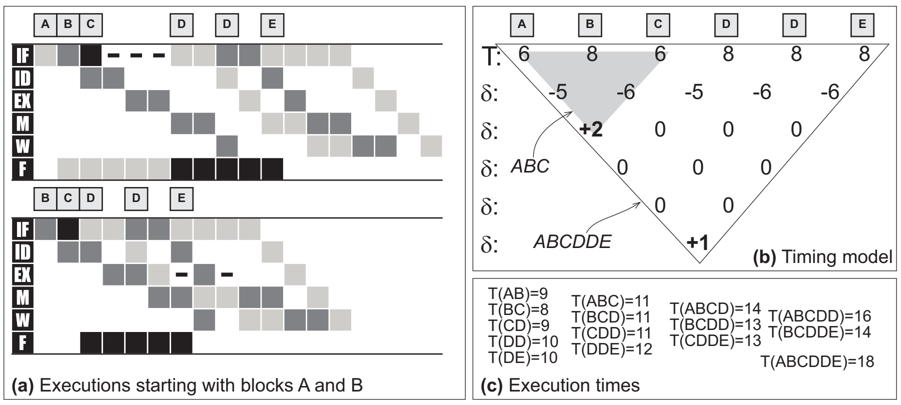


Figure 5.12: Example of double positive timing effect

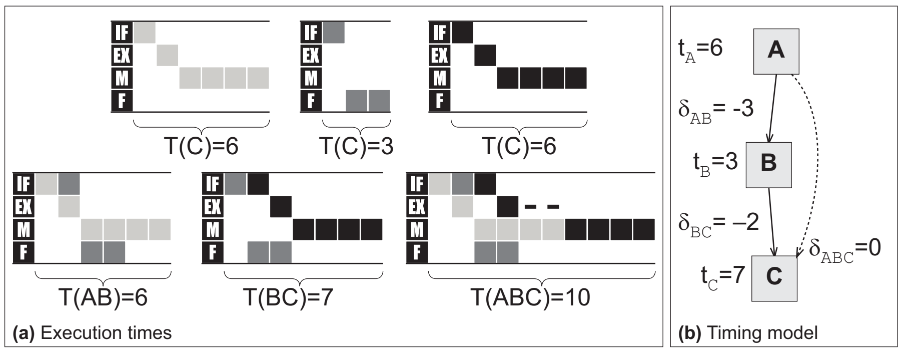


Figure 5.13: Example of no long timing effect despite pipeline interference

Parallel Pipeline Interference Must Cause a Positive Timing Effect When an instruction executing in one parallel pipeline interferes with another instruction in the same pipeline across a series of intermediate instructions that execute in some other pipeline (like the cases shown in Figure 5.12 and Figure 4.5), the result is not necessarily a positive timing effect. Depending on the time required for the intermediate instructions, the long timing effect might not materialize at all, as shown for example in Figure 5.13.

5.3 Constructing a Safe Timing Model

Our approach to low-level timing analysis is based on using general-purpose cycle-accurate simulators or actual hardware to generate the timing model for a program, as discussed in Chapter 4 and Chapter 6. The advantages of this approach is that retargeting to a new processor is very quick since no special-purpose model has to be constructed, and that the model potentially is exact. The disadvantage is that it is harder to ensure the safety of the analysis, since we cannot inspect the internal state of the pipeline and thus cannot precisely know if there are long timing effects waiting to happen.

The core problem to be solved for the analysis presented in Chapter 6 is to determine when a sequence of instructions starting from a certain instruction I_1 can cause more timing effects. If we are sure that I_1 cannot cause any more (positive) timing effects, we do not need to investigate longer sequences.

An easy way out is to reduce the precision by using a safe model, as discussed in Section 5.4.1 below. However, even precise models can be used safely in some circumstances.

5.3.1 Well-Behaved Single Pipelines

As shown in Theorem 5.5, if data dependences only appear between neighboring instructions, no positive timing effects can occur in a single pipeline. For such a pipeline, any timing model is safe, and analyzing longer sequences will just yield a tighter WCET estimate.

If all data dependences in a processor only reach from a stage j to its predecessor stage $j-1$, no data dependences can occur between non-adjacent instructions. This is thanks to the fact that since the dependence goes at most two points back in the pipeline (from point p_i^{j+1} to p_{i+1}^{j-1}), all dependences beyond the non-adjacent instructions will be subsumed by the regular constraints (as illustrated in Figure 5.2(c)).

In practice, this is a rather common case, since most pipelines have data forwarding paths [HP96] that avoid most data dependences, and in particular the long data dependence used in the example in Figure 1.7. For a pipeline like Figure 1.7 with full forwarding, data dependences only result from memory load instructions (that necessarily generate their data in the `MEM` stage) loading data that is needed by an immediately following arithmetic instructions (which is waiting to enter the `EX` stage).

Examples of processors with pipelines exhibiting this nice behavior are the ARM7 [ARM95] and ARM9 [ARM00b], NEC V850 [NEC95] (not the V850E), Hitachi SH7700 [Hit95], and Infineon C167 [Inf01].

5.4 Safety of Other WCET Analysis Methods

Although the discussion in this chapter uses the timing model in Chapter 4, the results are applicable to *all* WCET analysis methods. The occurrence of a long timing effect from an instruction I_1 to some later instruction I_m ($\delta_{I_1 \dots I_m} \neq 0$ in our model) indicates that the execution of I_1 in the processor pipeline causes a delay or speedup for instruction I_m. All pipeline analysis methods have to account for such effects in some way, even if they do not use our pipeline timing model.

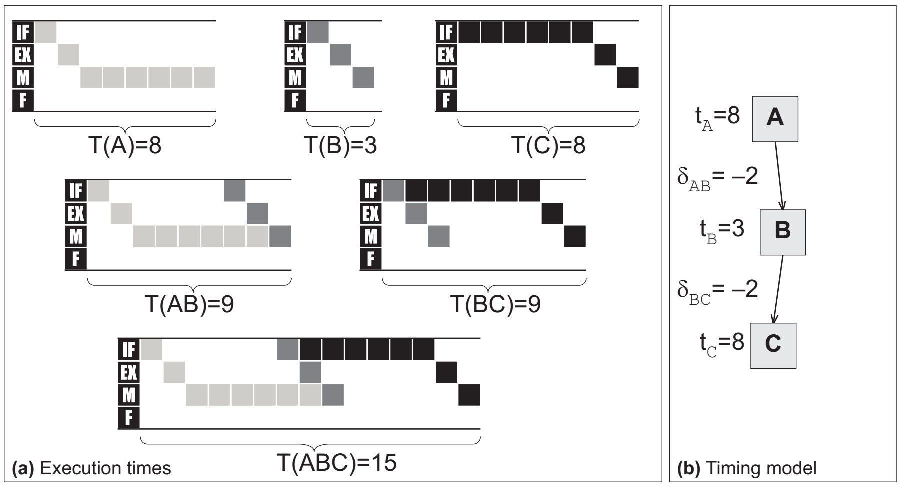


Figure 5.14: Pairwise conservative model of the scenario in Figure 4.8

5.4.1 Pairwise Conservative Analysis

Even if a certain pipeline structure (processor) is subject to long timing effects, it might be possible to construct a *model* of it that is not subject to long timing effects. Such a model would overestimate the execution times when long timing effects could occur, ignoring the potential speedup from long negative timing effects and taking the penalty from long positive timing effects early. The measurements discussed in Section 9.5.1 give an indication of the precision that could be lost if long timing effects are thus overapproximated.

One way to obtain a safe model is to only model the effect between pairs of adjacent instructions, and make sure not to model the potential speedup from overlap across more than pairs of instructions. In essence, this means working with "rigid" basic block models, as illustrated in Figure 5.14: the pipeline shape of all nodes is maintained, and thus the real-world overlap between `A` and `C` is not modeled, leading to an overall pessimistic (but safe) execution time for the sequence `ABC` of 15 cycles.

For parallel pipelines, we need to follow the rules of Lim et al. [LBJ$^+$95], where adding a new instruction to a sequence of instructions always adds at least one cycle (otherwise, we could get long timing effects). Figure 5.15 shows how the case in Figure 4.5 would be modeled: node `A` would not be allowed to completely overlap node `B` (as shown for the sequence `AB`), which makes the timing effect between the nodes minus four instead of minus five. Thus, the positive timing effect of the interference between `A` and `C` is taken early, on the edge between `A` and `B`, which is safe but pessimistic (ideally, the effect should only be counted when all the nodes `ABC` are executed in sequence, not when only `AB` are executed). Such models will also miss the speedup observed in Figure 4.6.

If a processor has instructions that can execute for quite a long time in parallel to other instructions, such a pairwise model is likely to give very high

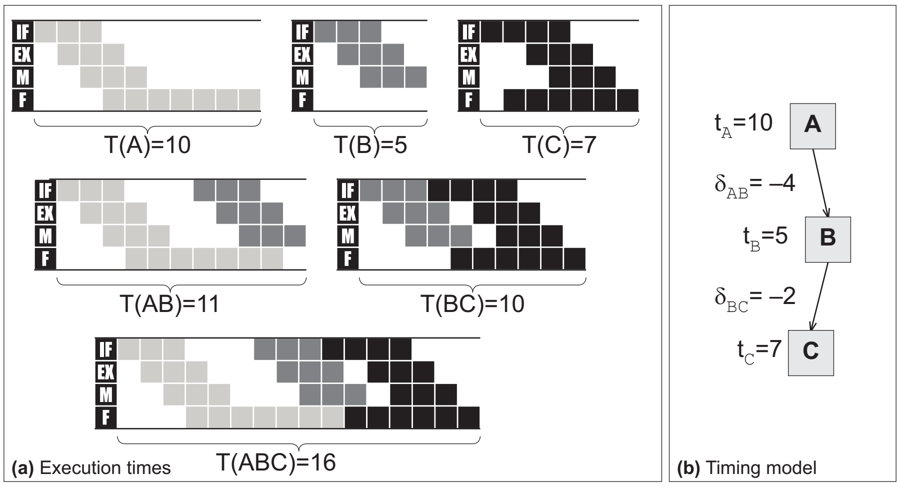


Figure 5.15: Conservative model of the case in Figure 4.5

overestimations. This is due to the fact that the overlap between a long-running instruction and the instructions or basic blocks after the immediate successor cannot be accounted for.

The safety of these models using rigid instructions can be derived from Theorem 5.6. Since an instruction is not allowed to change its shape, no stalls will occur, and thus no long timing effects will happen.

Note that our timing model and the timing analysis presented in Chapter 6 can use any hardware model, including a safe model like those presented here, and thus we can get a safe WCET analysis at the cost of some precision.

Data dependences between non-adjacent instructions are still a cause for concern for the pairwise models, since they can cause stalls that cannot be discovered without bringing the two instructions involved in the dependence together.

In the simplest case, pairwise models do not maintain an approximation of the pipeline state, only a number of cycles per instruction. Examples of this type of model are our ARM9 model described in Section 7.2, the C167 model created by Atanassov et al. [AKP01] (see Section 8.2.3), and the execution time model for Java byte codes by Bate et al. [BBMP00]. The C167 model in particular has been validated to be safe visavi the hardware, and is documented as sometimes overestimating the execution time [AKP01]. In these particular examples, no long data dependences can occur, and the analysis is thus safe.

A safe model incorporating a detailed pipeline model is the pairwise reservation-table approach used by Lim et al. [LBJ$^+$95] and Colin and Puaut [CP01c]. Lim et al. note that they handle data dependences, but do not discuss the details of the modeling.

The approach by Stappert and Altenbernd [SA00] uses reservation tables on a basic-block level to model an out-of-order superscalar PowerPC 604 processor.

Only pairs of basic blocks are considered, and if the reservation tables and concatenation functions used are safe, this could result in a safe analysis. Due to the more complex hardware modeled, we cannot determine whether the model is actually safe or not (and as noted in Chapter 8, it is very hard to build an accurate model for a processor as complex as the PowerPC 604).

We also like to point out that building safe models for WCET estimation has the small drawback that another hardware model has to be defined in order to perform BCET analysis. A model that is safe for WCET analysis is by necessity unsafe for BCET analysis, since the WCET model overestimates the execution time in cases of uncertainty, while a BCET model would need to underestimate the time in such cases.

5.4.2 Path-Based Low-Level Analysis

A path-based low-level analysis examines complete paths through some section of a program, instead of just pairs of basic blocks. A path can be formed across multiple basic blocks, and thus timing effects across several instructions will be accurately accounted for within each path. The disadvantage of this approach is that the number of possible execution paths can potentially be very large for complex code. Note the distinction between a path-based *calculation* and a path-based *low-level analysis*. For example, it is posssible to use a path-based low-level analysis with an IPET calculation.

In most cases, paths will not extend across the entire program execution, and thus at some point, two paths P_1 and P_2 have to be appended to each other to compute the execution time for larger sections of a program. Long timing effects between instructions in P_1 and P_2 will not be accounted for, since no analysis is performed across the boundary between P_1 and P_2. Thus, the append must be performed in a safe manner. If the pipeline used has the potential for positive long timing effects, we must use an empty pipeline assumption at the append point. Unfortunately, the result can be a rather large loss in precision, since we cannot account for the pipeline overlap between the successive paths. If only negative long timing effects can occur, some estimate of the potential overlap between P_1 and P_2 should be performed in order to increase precision.

At end of a section analyzed using complete paths (like the end of a loop body), the pipeline state of all the possible paths for the section have to be merged, before the merged state is used to append to the path for the following section. At such merge points, it is critical that potential long timing effects are taken into account, or else we risk underestimating the WCET.

The pipeline analysis designed by Healy et al. [HAM$^+$99] is a path-based approach that uses reservation tables concatenated along paths. They have seen pipeline effects across several instructions in their experiments, due to the slow floating point unit of the MicroSPARC I, which would allow as many as fifty integer instructions to execute in parallel to a single floating point instruction. When appending paths, they compute the overlap, but since their processor is potentially subject to positive long timing effects from the parallel pipeline, this is not necessarily safe. However, they report no cases of underestimation in their experiments. At merge points, they are slightly conservative and compute some form of union of the pipeline state for all the paths.

Ziegenbein et al. [ZWR$^+$01] also execute long paths in the program on a simulator or actual hardware, and use empty pipeline assumptions between paths. This is safe, and results in a certain overestimation, as reported in their experiments.

It would be interesting to consider a combination of our timing model and low-level path-based timing analysis, working with paths instead of nodes as units in the timing model. The nodes currently used can actually be considered as short paths, since they usually contain several instructions that are analyzed as a unit. We would expect that we would get fewer long timing effects, since we are increasing the size of the units in the analysis. On the downside, we would probably get more units to analyze, since the number of paths in a program segment can be rather large.

5.4.3 Abstract Interpretation-Based Analysis

The abstract interpretation-based models of Schneider et al. [SF99, FHL$^+$01] are claimed to provide safe models of the behavior of processor pipelines. Several possible pipeline states are maintained for each point in the analysis, which should mean that they also propagate pipeline states across more than neighboring instructions. Thus, they gain in precision but also need to consider long timing effects since they are not performing a conservative pairwise analysis.

It is not clear if there is a limit on the number of states propagated, since this would seem to be a potentially rather large set. If some limit is enforced, the reduction of the set of states has to be carefully designed to make sure that all stalls that have not yet manifest themselves as timing effects are considered.

Chapter 6

Timing Analysis

This chapter presents the *timing analysis method* we have designed in order to generate the concrete values of the timing model described in Chapter 4. As illustrated in Figure 6.1, this chapter presents the analysis part of the separation between model and analysis.

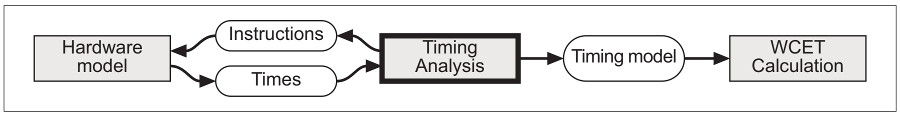


Figure 6.1: Isolating the timing model from the hardware model

6.1 Algorithm

The input to the timing analysis algorithm is the timing graph, containing nodes pointing to basic blocks in the program code and execution scenarios describing how the basic blocks execute. The timing analysis uses Equations (4.1) and (4.2) to generate node times and timing effects from the raw execution times returned by the timing function T.

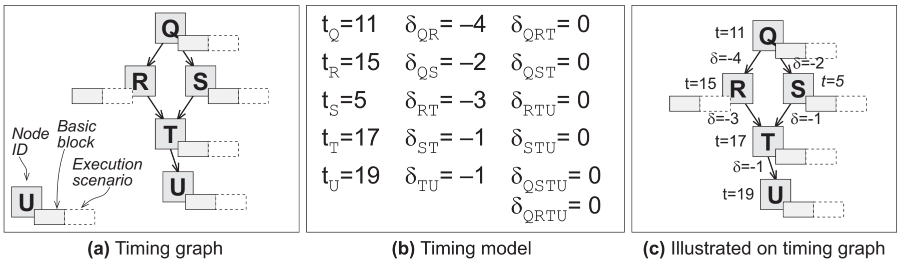

Q
R
S
T
U
Node ID
Basic block
Execution scenario
U
(a) Timing graph
$t_Q=11$ $\delta_{QR}=-4$ $\delta_{QRT}=0$
$t_R=15$ $\delta_{QS}=-2$ $\delta_{QST}=0$
$t_S=5$ $\delta_{RT}=-3$ $\delta_{RTU}=0$
$t_T=17$ $\delta_{ST}=-1$ $\delta_{STU}=0$
$t_U=19$ $\delta_{TU}=-1$ $\delta_{QSTU}=0$
$\delta_{QRTU}=0$
(b) Timing model
t=11
δ=-4
δ=-2
t=15
t=5
δ=-3
δ=-1
t=17
δ=-1
t=19
(c) Illustrated on timing graph

Figure 6.2: Pipeline analysis and modeling

As illustrated in Figure 6.2, the timing analysis results in a timing model for the program. The timing model can be illustrated by adding the time values for nodes and timing effects for edges to the timing graph.

```
TimingAnalysis(TimingGraph $TG$, TimeFunction $T$):
  /** Local variables **/
  TimingModel $M$ /** Mapping from sequences to times **/
  TimeDatabase $D$
  SequenceQueue $Q$
  Sequence $s$
  Integer $l$
  /** Run all nodes **/
  for each node $n$ in $TG$ do
    $s$ = $\{n\}$
    add $T(s)$ to $D$ /** Here we run the sequence **/
    $t_n$ = $D$.time($s$)
    add <$n$,$t_n$> to $M$
    for each successor $m$ to $n$ in $TG$ do /** Add all sequences of length two**/
      add $s \cdot m$ at end of $Q$ /** Append $m$ to sequence $s$ **/
    end for
  end for
  /** Investigate successive sequences **/
  while $Q$ not empty
    $s$ = head of $Q$ /** $s$ is removed from $Q$ **/
    $l$ = length($s$)
    add $T(s)$ to $D$
    $\delta_s$ = $D$.time($s$) - $D$.time($s[1..l-1]$) /** $s[x..y]$ is a subsequence of s **/
        -$D$.time($s[2..l]$) + $D$.time($s[2..l-1]$)
    add <$s$,$\delta_s$> to $M$
    if $Extend(s)$ then
      for each successor $m$ to $n$ in $TG$ do
        add $s \cdot m$ at end of $Q$
      end for
    end if
  end while
  /** Return the result **/
  return $M$
```

Figure 6.3: Timing analysis algorithm

We assume that we have access to a hardware model (timing function) that is opaque, i.e. we cannot investigate the internal state of it. A typical example of such a timing function is a cycle-accurate trace-driven simulator for a CPU, that is, a CPU model that returns a cycle count given a sequence of instructions, but that does not consider the values computed and the contents of registers and memory. In the case that instruction semantics has an effect on the execution time (like instructions with input-dependent execution times or memory accesses via cache memory hierarchies), the execution scenarios must provide enough information to determine the execution time. Note that we need an interface that allows us to provide the timing function (processor simulator) with the information in the execution scenarios.

A pseudo-code description of the timing analysis algorithm is shown in Figure 6.3. We start by generating times for all nodes, and then all pairs of nodes are entered into the queue (Q) of sequences to analyze. For each sequence s in the queue, we get the execution time from the time database (D.`time(`s`)`) and calculate the corresponding timing effect. The time database (D) stores all times obtained from the timing function, and runs a sequence through the timing function only if the database contains no time for it. Thus, no sequence is ever run more than once in the hardware model. Since the hardware model can be rather complex and have a large execution time cost, this helps bring down the absolute run time of the algorithm.

A central part of the algorithm is the *extension condition*, a function that determines whether we have found all positive timing effects along a certain sequence. If positive timing effects are judged to still be possible, all the possible extensions to the sequence are entered into the queue Q. It is also nice if the extension condition allows us to find as many of the negative timing effects as possible, since this will increase the precision of the analysis. More details are given in Section 6.3 below.

6.2 Using the Hardware Model

To reduce the coupling between components and thus enhance retargetability, we do not want to build a special-purpose model of each target processor within the timing analysis module. Instead, as explained in Chapter 3, we use a separate hardware model (timing function) which is supposed to be opaque, but still controllable enough to be useful. The requirements on the model are that it must be possible to control the hardware model enough to enact the effects of the execution facts, and that the model is deterministic so that the same sequence of instructions and execution facts always give the same execution time. This should be possible to achieve using a standard processor simulator with some interface code.

Note that the same code can take different time to execute in a pipeline if the initial states of the pipeline are different (different preceding instructions). The effect of the differing initial conditions between program nodes is captured by the timing effects, which accurately model the fact that a certain node might have different execution times depending on its predecessor in the pipeline (consider the execution of block `C` in Figure 4.5 and block `B` in Figure 4.8). Thus, only the initial state of the program to be analyzed can have an effect on the overall execution time. The initial state can typically be assumed to be a cold pipeline, and the impact of a mistake in this assumption is at most a few cycles at program start, which can be accounted for by a safety margin.

6.3 Extension Condition

The extension condition is at the heart of our pipeline analysis method, and defining the extension condition is the central problem when porting our pipeline analysis method to a new CPU (or more precisely, to a new hardware model).

The goal is to formulate a condition that is *efficient*, in that it evaluates as few unnecessary sequences as possible, and *safe*, in that it never misses a positive long timing effect (since we could otherwise underestimate the WCET). It must also work with the information available to it by the principle of separation of components, which we assume does not include the detailed internal state of the hardware model.

In the following, we discuss the extension conditions that we have designed and used in our experiments.

6.3.1 Simple Extension Condition

For a processor or conservative processor model where no long timing effects can occur, the extension condition is simply to return `false` for all sequences of length two or longer (no need to analyze *beyond* length two).

$$\texttt{extcond-simple} \triangleq (length(s) < 2)$$

This condition is safe and precise for pipelines that satisfy Theorem 5.2 or Theorem 5.3. It is also applicable for pipeline models that are conservative in the fashion discussed in Section 5.4.1.

6.3.2 Extension Condition for Scalars

For more complex pipelines, we propose an extension condition based on the observation that most long timing effects occur close to when an instruction enters the pipeline. Thus, we select to stop looking for more timing effects when all instructions of the first node in a sequence have left the pipeline.

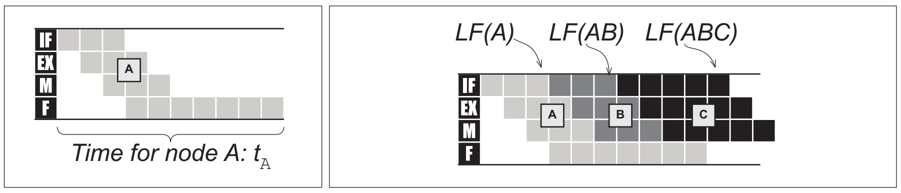
Considering our assumption of an opaque hardware model, we cannot directly observe the instructions in N_1 as they execute, but we can observe the following:

- The execution time for a sequence of instructions, which is the value of the timing function $T(N_1 \ldots N_n)$.
- The last time an instruction is fetched (enters the pipeline) for a sequence of instructions, which can be determined since we feed the timing function with instructions. The last fetch of an instruction in a sequence of nodes $N_1 \ldots N_n$ is called $LF(N_1 \ldots N_n)$. Figure 6.4 shows the last fetch times for three progressively longer sequences: `A`, `AB`, and `ABC`.

Given the T and LF functions, we define the extension condition for a sequence of nodes $N_1 \ldots N_n$ as:

$$\texttt{extcond-scalar} \triangleq (LF(N_1 \ldots N_n) \leq t_{N_1})$$

The extension condition `extcond-scalar` returns `true`, i.e. commands the analysis to consider extensions of the sequence $N_1 \ldots N_n$, as long as any of the instructions from the first node in the sequence (N_1) are still in the pipeline.



(a) Node A in isolation **(b)** Sequence ABC with last fetch times marked

Figure 6.4: Illustration of last fetch times

For a pipeline without positive timing effects, like those discussed in Section 5.3.1, using this extension condition will yield a safe but potentially imprecise model (since some really long negative timing effects could conceivably be missed).

For pipelines with positive timing effects that only take effect immediately (i.e. when the stall causing the timing effect occurs), this extension condition should also be safe, since it waits until all instructions in the first node has left the pipeline and thus all stalls have manifest themselves.

6.4 Complexity

The goal of the pipeline model and analysis presented in this thesis is to create an efficient WCET analysis by avoiding the need to examine long paths (sequences) in the program (the number of which is exponential in general). Thus, the complexity of the pipeline analysis method should be investigated.

Let f be the greatest fan-out of the timing graph, i.e. the number of successors to a node. Then the worst-case number of sequences that need to be investigated is nf^{m-1}, where n is the number of nodes in the timing graph and m is the length of the longest sequence investigated.

However, this complexity is often much larger than that which is actually encountered in practice, since the fan-out for a typical program is on average much less than f. A branch statement has a fan-out of two, while most nodes have a single successor. `switch`-statements or calls to function pointers can give rise to very high local fan-outs, but they are very few compared to the total number of nodes in the program. Furthermore, not all sequences up to length m need to be investigated; only those where there is a risk of encountering long timing effects. To bring the complexity down, the extension condition needs to be more selective than just looking at the length of sequences.

Measurements indicate that in general, the analysis is linear in the size of the program, as shown by Figure 9.5 and Figure 9.9 in Chapter 9.

Chapter 7

Applications of the Analysis

We have applied the analysis described in Chapter 6 to two concrete pipelines: the NEC V850E and the ARM9. This chapter describes the details of those applications.

7.1 The NEC V850E

The NEC V850E [NEC99] is the second-generation core in the NEC V850 family [NEC95]. The instruction set architecture (ISA) is an extension of the V850, and the pipeline has been extensively redesigned to improve performance. Microcontrollers based on the V850E core are available up to a speed of about 100 Mhz; with typical clock speeds between 33 and 66 Mhz. A variety of on-chip peripherals are available, depending on the particular model.

The main performance-improving factor of the V850E compared to the V850 is the "additional" pipeline shown in Figure 7.1. This allows branches, loads, and stores to execute in parallel to regular instruction execution in the main pipeline. The NEC V850E also allows for *pairs* of instructions to start executing in parallel; for example, a short load (`SLD`) and short arithmetic instructions (encoded in 16 bits) can be issued on the same cycle, if they are present within the same 32-bit word [NEC99, Mon00]. This behavior is dictated by the instruction layout in memory: only aligned pairs can be issued in parallel, and so the V850E does not really qualify as a "superscalar" CPU. There is a simple

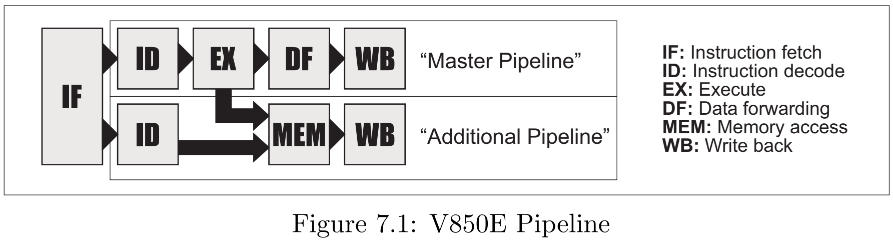


Figure 7.1: V850E Pipeline

instruction prefetcher to increase performance.

The memory architecture is Harvard style, with separate data and address buses for for instructions and data [HP96]. Typical microcontrollers based on the V850E core feature on-chip single-cycle ROM (or FLASH) for programs, and SRAM for data, making the timing behavior quite easy to analyze.

7.1.1 Hardware Model

The V850E hardware model used is a cycle-accurate trace-driven simulator modeling the pipeline in great detail. It has been developed within our research group to gain insight in how a simulator works and to facilitate experimentation by affording us complete control over the simulator.

The simulator has been validated against real hardware for the case that we have single-cycle memory for both instructions and data [Mon00]. Each run of the processor (when used as a timing function for WCET analysis) is assumed to start from an empty pipeline state, with an empty prefetch queue. Some experiments were performed using longer data memory latencies, which made it possible to generate long timing effects. The behavior in this case has not been validated against hardware.

7.1.2 Extension Condition

For the V850E, we have used the extension condition `extcond-scalar`, described in Section 6.3.2. In experiments, `extcond-scalar` did successfully detect all timing effects, as discussed in Section 7.1.3.

7.1.3 Long Timing Effects

In our main line of experiments with the V850E, with single-cycle data memory, no timing effects across longer sequences than two nodes were observed. However, we did manage to provoke long timing effects by increasing the data memory access time to six cycles. In this case, timing graph nodes containing data memory accesses had the potential to overlap the following nodes, potentially generating long timing effects.

In the experiments (as reported in Section 9.5) only negative long timing effects were observed (created in the fashion shown in Figure 4.4). Despite this experimental result, it should be possible to get positive long timing effects on the V850E. By manually writing assembly code, we found that the sequence of instructions "`STW-BR-STW`" does generate a positive timing effect, in the manner shown in Figure 4.5. However, no case of positive timing effect was observed in our experiments.

There are several properties of the V850E that conspire to generate only negative timing effects in our experiments. The most important factor is that perfect instructions[1] executing between memory accesses do not cause positive

[1]As defined in Section 5.2.1.

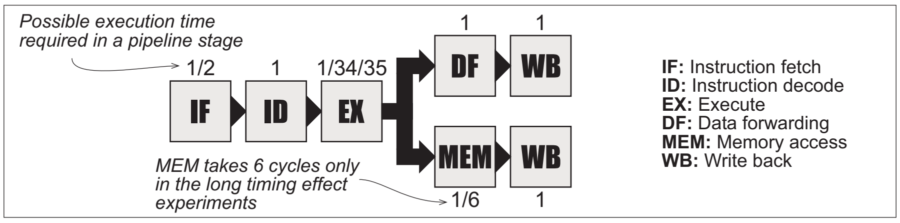


Figure 7.2: V850E Pipeline with execution times for each stage

long timing effects, thanks to the particulars of the V850E pipeline. This is a real-life example of the case shown in Figure 5.13, where a disturbance due to a parallel pipeline does not manifest as a long timing effect.

Perfect instructions are very common, since almost all instructions only require a single cycle in each pipeline stage. Figure 7.2 shows a reduced diagram of the V850E with the possible number of cycles in each stage (due to the special behavior of the additional pipeline, this diagram shows a pipeline organization that is equivalent to that in Figure 7.1). The notation x/y means that instructions use either x or y cycles in the pipeline stage. In the `IF` stage, only misaligned branch targets and the `MOV32` instruction use more than one cycle, and the only instruction using more than one cycle in `EX` is division. Hence, almost all instructions that do not access memory are perfect instructions.

Furthermore, data dependences are of the well-behaved kind discussed in Section 5.3.1, which means that each of the Master and Additional pipelines in themselves cannot generate positive long timing effects. Thus, only the effects of the parallelism can cause long timing effects.

Note that since we are working with *nodes* containing several instructions, many effects will never be seen since they are internal to nodes. Working on the node level reduces the practical complexity of the WCET analysis.

Considering the applicability of the pipeline model in Chapter 5, we note that when investigating various instruction sequences on the V850E, the model and "reality" (in the form of the V850E simulator) consistently gave the same timing results.

7.1.4 Miscellaneous Notes

On the V850E, as on most pipelined processors, conditional branch instructions have different execution times depending on whether the branch is taken or not. Taken branches require more clock cycles to execute. Thus, for sequences of nodes ending with a conditional branch instruction, we assume that the branch is taken. For branches in the middle of a sequence, the time taken for the branch depends on whether the next instruction corresponds to the taken or not-taken cases of the branch, as illustrated in Figure 7.3.

In our implementation, we use execution facts that are conditional on the context in which they are used. Such facts contains a number of pairs `(nextnode,fact)`, and a default case. This removes the need to decorate the

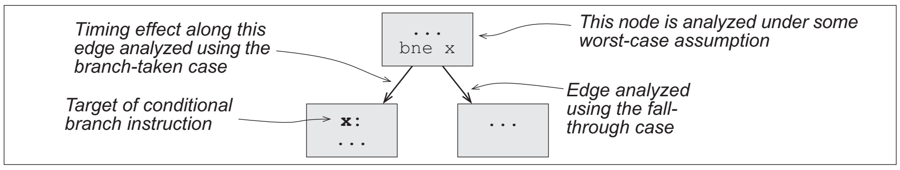

Timing effect along this edge analyzed using the branch-taken case
...
bne x
This node is analyzed under some worst-case assumption
Target of conditional branch instruction
x:
...
...
Edge analyzed using the fall-through case

Figure 7.3: Modeling a conditional branch

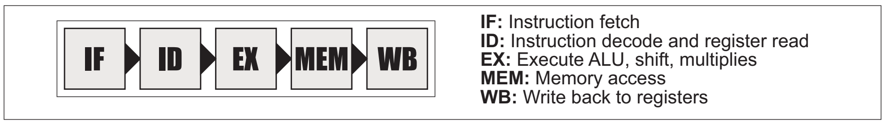

IF
ID
EX
MEM
WB
IF: Instruction fetch
ID: Instruction decode and register read
EX: Execute ALU, shift, multiplies
MEM: Memory access
WB: Write back to registers

Figure 7.4: ARM9 Pipeline

edges in the timing graph with facts, which would have been the alternative implementation.

7.2 The ARM ARM9

The Advanced Risc Machines Ltd. (ARM) ARM9 [ARM00b] was launched in late 1997. It is an integer-only processor core which can be licensed for use in custom integrated circuits (ASICs). ARM also supplies some macrocells, combining the ARM9 core with caches, to make it simple to create complete microcontrollers based on the ARM9. ARM9 cores are typically configured to run between 100 Mhz and 200 Mhz, but the exact clock speeds available depend on the manufacturing process used [Tur97].

The five-stage pipeline of the ARM9 core is shown in Figure 7.4. The memory interface is Harvard, with separate address and data buses for instructions and data. Typically, an ARM9 core would be equipped with separate instruction and data caches, although it is conceivable to implement an ARM9 core with SRAM for instructions and/or data instead of caches (this is actually recommended by ARM for real-time systems [ARM01]).

ARM recently released an update to the ARM9, the ARM9E [ARM99, ARM01], which includes some DSP extensions in the instruction set and a much faster multiplier circuit. We modeled only the basic ARM9.

The documentation from ARM [ARM00b] is quite scarce on the layout of the pipeline of the ARM9. The diagram in Figure 7.4 is taken from [Tur97], and a very similar pipeline structure is used in the ARM9E [ARM99].

7.2.1 Hardware Model

Our ARM9 model was described to support a specific project involving WCET analysis on the interrupt disable regions of the OSE Delta real-time operating system [Car02].

The hardware model used in our implementation for the ARM9 is a simple table of execution cycle count for each instruction, based on the types of the operands and the operation, with additional rules for when data dependences delay the instruction execution. The model is based on Table 7.2 in the ARM9 user's manual [ARM00b], and has not been verified against real hardware. All instructions with execution time depending on the value of operands unknown at compile time, like the multiplication instruction, are assumed to execute for the longest time possible. Since no effects across more than two instructions are documented in the manuals, we assume that this is a safe model in the manner discussed in Section 5.4.1 (the pipeline organization supports this, since data depedencies can only occur from `MEM` to `EX`, and thus only between adjacent instructions).

For our experiments with the ARM9, we assumed a perfect memory system with single-cycle access to all instructions and data. No caches were modeled.

The initial condition for each run of the hardware model is an empty pipeline with no data dependences coming in, since we assume that a program starts without an incoming register-write dependence (which is very reasonable, since the instruction leading to the execution of the program usually is some form of branch).

We were considering using the cycle-accurate ARMulator ARM simulator from ARM Ltd., but we could not find any documentation on how to use it as a backend (the documentation deals with how to plug in new memory, cache, and peripheral models into the ARMulator, but not on the interface for running code in the ARMulator itself). Futhermore, the cost of a license was prohibitive.

7.2.2 Extension Condition

We used the extension condition `extcond-simple` for the ARM9 experiments, since the hardware model was a safe pairwise model.

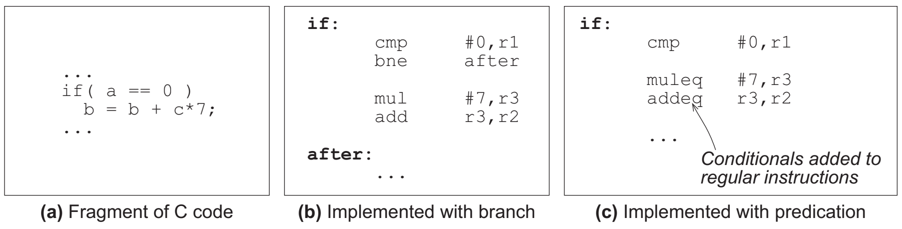
7.2.3 Long Timing Effects

Since the processor model used is a safe model according to Section 5.4.1 and we used the extension condition `extcond-simple`, no long timing effects were observed.

7.2.4 Miscellaneous Notes

An interesting feature of the ARM processors is the use of *predicated execution.* Predicated execution means that regular computational instructions can be conditional, with their results ignored if some condition is not met. In traditional instruction set design, only branches are conditional, and pieces of code are skipped by jumping past them. In an architecture with predicated execution, short jumps can be avoided by making the instructions to be skipped conditional. The concept is illustrated in Figure 7.5: contrast the jump in

Figure 7.5(b) with the predicated instructions in Figure 7.5(c). Extensive predicated execution is present on the ARM, HP-PA, and Intel/HP Itanium architectures. Many other architectures support it in a limited form with a conditional move instruction, like the NEC V850, Motorola/IBM PowerPC, and Sun SPARC.



```
...
if( a == 0 )
  b = b + c*7;
...
```

(a) Fragment of C code

```
if:
        cmp     #0,r1
        bne     after

        mul     #7,r3
        add     r3,r2

after:
        ...
```

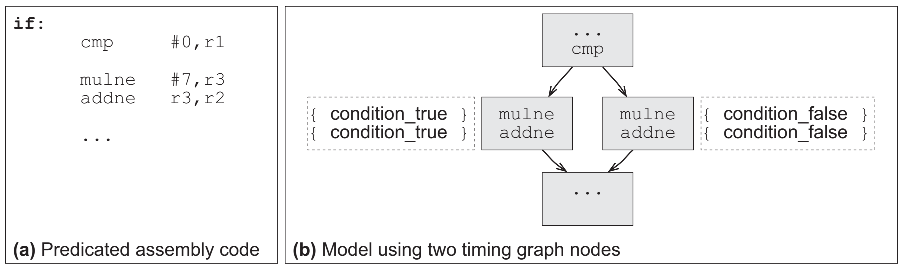
(b) Implemented with branch

```
if:
        cmp     #0,r1

        muleq   #7,r3
        addeq   r3,r2

        ...
```

(c) Implemented with predication

Figure 7.5: Example of predicated execution

To correctly account for predicated instructions in WCET analysis, we need consider their pipeline behavior. In many cases, the timing and pipeline behavior of a predicated instruction is the same no matter if the condition is true or not, and the only difference is whether the result of the operation is actually recorded in the processor's register file. When the timing or pipeline behavior is different (for example, a predicated load instruction that only accesses memory when the condition is true, or a time-consuming division instruction that just skips its calculation when the condition is false), we can either use a safe worst case (typically executing the instruction), or introduce a separate timing graph node for each case, as shown in Figure 7.6(b).



```
if:
        cmp     #0,r1

        mulne   #7,r3
        addne   r3,r2

        ...
```

(a) Predicated assembly code

(b) Model using two timing graph nodes

Figure 7.6: Duplicated timing graph nodes to model predicated execution

We saw no need to model the predicated execution of the ARM9 in this detail, since there is no explicit model of the pipeline state that could be different in the taken and not-taken cases. Unless instructed otherwise by execution facts, the ARM9 analysis assumes worst-case execution times for all instructions, which means that predicated instructions are assumed to execute.

Chapter 8

Building Hardware Models

Static WCET analysis relies on *models* of the processor on which a program is executed. For pipelined processors, a variety of modeling methods have been used: cycle-accurate simulators [EE99, ZWR$^+$01], special-purpose models using reservation tables or similar techniques to model the behavior of the processor pipeline [LBJ$^+$95, Sta97, HAM$^+$99, CP01c], dependence graphs [LHKM98], pipeline abstractions [SF99, FHL$^+$01], and tables of instruction execution times and inter-instruction effects [BBMP00, AKP01, Car02]. All approaches share the common problem of not using the actual hardware but rather a model of it, and thus bringing the quality of the model into question. This chapter will discuss the issues involved in building hardware models.

8.1 Error Sources in Model Construction

Considering the design flow for hardware models, several sources of errors can be identified. Figure 8.1 shows an overview of the design and implementation work separating a simulator from the physical chips shipped by the processor manufacturer.

Starting with the design for the chip, engineers implement the design as hardware, creating a program in VHDL (or other hardware description language) describing the implementation of the processor. This model is then compiled through a synthesis tool to generate a physical layout which is used to manufacture the actual processor chips. In parallel to the implementation work, manuals are written. The tool developer who is attempting to build a simulator for the processor reads the manuals, and creates the design for a simulator. The design is then implemented to create a simulator that can be run. In each step, errors can be introduced.

In the *hardware implementation* stage, errors can be introduced relative to the design. The most famous recent example is the Pentium `FDIV` bug [Pri95], where a bug was present in millions of Intel Pentium processors sent to cus-

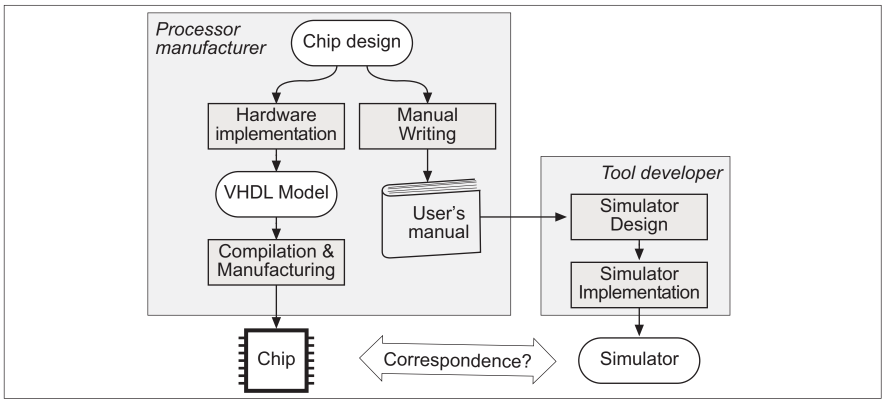


Figure 8.1: Schematic flow of simulator implementation

tomers. The presence of extensive errata lists from various processor vendors further show that hardware does contain bugs which are not found until after products have been shipped to customers and used in products [Int01b, Mot01]. However, it should be noted that correct behavior is the norm for the commonly used parts of an architecture. The bugs mostly relate to rarely-used features like performance counters.

Bugs related to the timing behavior of processors are more common, since timing is usually not tested and validated as extensively as function. The large number of bugs relating to interrupt timing (where sometimes desktop processors can lock up in unfortunate scenarios [Int01b]) indicate that timing correctness and timing bugs is a problem not only in real-time software design but also in hardware design. Note that a timing bug in the hardware is likely to be diagnosed as a manual-writing error by a simulator designer, since it will be observed as a discrepancy between the manual and the hardware.

The *hardware compilation and manufacturing* stage takes the design from the VHDL code (or other hardware synthesis language) to actual processors. Here, faults can be introduced by errors in the compilation process or manufacturing defects. This is however rather rare today, and the processors that are shipped to customers are usually free of defects caused by this stage.

Manual writing is a major source of errors and vagueness, since it adds a (necessary) layer of interpretation between the users and the processor specification, making the manuals probably the biggest source of errors. The manual writers have to interpret the processor design, and this interpretation can be different from that of the hardware implementors. Simple typos add to the confusion that can be created. Since the intended audience for the manuals is usually regular programmers using the processor, many of the details needed to implement a precise simulator (or an optimizing compiler) are abstracted away or glossed over. Rules of thumb and tips for code optimization are more important than details about the pipeline operation. Thus, the manuals typically have a rather tenuous relation to the processors they are supposed to document, and this presents a major hurdle for outside simulator implementors. The conventional wisdom among embedded system engineers is that hardware documentation is usually not to be trusted[1].

In the *simulator design* step, the abstractions used when modeling the processor are decided, and the manuals and other available information is scoured for details on how the processor works. Not using up-to-date manuals, or misinterpreting the manuals cause errors. Instruction timings have to be decided, and decisions have to be made on the level of detail in the model (with the risk that important details are overlooked or simplified too much). The model will be an abstract version of the real processor, but ideally, the abstractions used accurately capture the behavior of the processor concerning the execution time of instructions.

During *simulator implementation*, errors are introduced as the designed simulation model is converted into actual program code. All programmers make mistakes, and the implementation of a simulator is no exception to this rule.

Hence, we can see that the hardware models we are using in static WCET analysis are several bug-inducing steps away from the actual hardware we are trying to model. This is a very serious problem for safe WCET analysis, since even if all the analysis methods are correct, the errors introduced by the hardware model can still make the analysis unsafe. This means that it is very important to *validate* the hardware models used in WCET analysis.

Looking at Figure 8.1, it seems that it would be possible to shortcut the process by deriving the simulator directly from the VHDL code from the processor. However, using the VHDL code is not a feasible approach since it is in general a very closely guarded secret of a processor manufacturer. Furthermore, the analysis required to abstract from the very detailed level of VHDL to the instructions-and-clock cycles required for simulators is very complex and not feasible to automate. Running a VHDL-level simulator is far too slow for practical use, but is a good way to obtain a reference for validating processor simulators.

8.2 Validating Simulators: Experience Reports

Most computer architecture research and design work is performed using simulators of various kinds [BC98, DBK01], and some research groups and commercial teams have reported their experiences in validating processor simulators against the real hardware. Here, we will summarize the reports to gain an understanding of the difficulties in building accurate processor simulators. We spend some

[1]Schneider and Ferdinand [SF99] note that the manual for the SuperSparc I processor they are modeling by abstract notation is deficient, and that they make pessimistic assumptions to compensate. Unfortunately, they do not explain how their safe pessimistic model was constructed, and how they can know that their model is pessimistic against the unknown information not contained in the manual.

extra space on the V850E case study, since we have good insight into it and it provides several interesting observations.

8.2.1 V850E Model

Montán [Mon00] validated and debugged the simulator we use for the V850E (see Section 7.1) by comparing it with a hardware implementation. The hardware used was a V850E emulator from NEC. An emulator is a special processor intended to help software developers debug their code by providing a more powerful debug interface than a regular processor. Using an emulator made it much easier to obtain execution times, but raises the question of the relation to the real hardware. According to NEC, exactly the same processor core is used in the emulator and the processors, the only difference being in the packaging. The validation work was initiated after we realized that the quality of the hardware model is a critical correctness component of a WCET analysis tool.

Methodology

The testing methodology employed for validating the V850E simulator is illustrated in Figure 8.2.

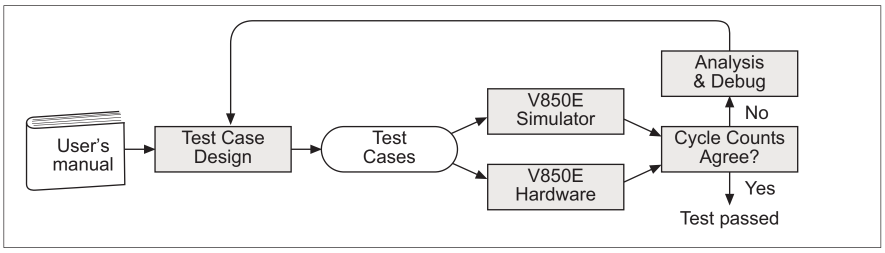


Figure 8.2: Work flow of the validation process for the V850E

The first step was to perform a manual analysis of the instruction set of the V850E and the pipeline behavior of the instructions, using information from the manuals [NEC99]. The result of the analysis was a set of test cases, each consisting of a few instructions. In several instances, improvised tests on the hardware was necessary to clarify the behavior of instructions when the manuals were vague or contradictory, adding an exploration element to the basic analysis.

The test cases strived to cover all relevant behaviors of the V850E with a minimal number of tests using a sophisticated classification of instructions into equivalence classes. To get precise timing measurements on the hardware, each test code sequence was repeated about one thousand times. A prologue and epilogue of `NOP`s was added to the instruction stream. Test sequences containing branches were constructed in such a way that the branches were always or never taken, despite the repetition of the code sequence.

The prologue and epilogue were run on the simulator and the hardware, and the difference was used to calibrate the measurements, since the hardware and simulator used different criteria for when to start and when to stop timing a code sequence.

For each test, cycle counts were compared, and if a difference was found, the test was analyzed in detail, trying to find the fault in the simulator that caused the discrepancy. In some cases, this process generated new test cases, since more information was needed to find the bugs, or to clarify the behavior of the hardware (the feedback loop in Figure 8.2). The information from the manuals was in effect extended with knowledge gained from the hardware; in many cases, errors were found in the manuals.

Classes of Bugs

The bugs found in the validation process were related to all error sources discussed in Section 8.1. The simulator was originally based on an outdated version of the manual, leading to the additional pipeline (see Figure 7.1 in Section 7.1) not being part of the model, which gave an initial discrepancy of up to 30% between the simulator and the hardware. This demonstrates the need to keep up-to-date with documentation updates.

After this correction, the error was much smaller, but still many more bugs were found, some of the more interesting of which are summarized in Figure 8.3. Note that the validation produced evidence of a hardware bug, which was independently found by NEC. This shows the need to be very careful when examining hardware, and that hardware bugs can slip past manufacturer testing.

Improvement in Accuracy

We used exact clock cycle agreement as the criterion for passing a test, and as the validation effort progressed, more and more of the small test cases passed.

To validate that the simulator was actually improving in accuracy on real programs as well, a number of test programs were selected and run in the simulator and on the hardware, for each version of the simulator. The results are shown in Figure 8.4.

For most programs, the final error is down to about 10 cycles, which can be explained by differences in measurement start and stops between the hardware and the simulator (note that the `NOP` technique was not applied to the large test programs). This makes the percentage error rather meaningless, since it is larger for programs with shorter execution times. However, for `matmult` and to a lesser extent `jfdctint`, there are indications of residual errors in the simulator. For `matmult`, most of the error is due to the fact that our simulator did not model the missing interlocks between successive `SLD` instructions.

In total, the work took about five months, and resulted in a simulator which generates execution times which are very close to the timing of the real V850E processors. There are some remaining differences, especially the timing of the buggy `SLD` instruction.

Manual Writing
`NOP` instruction documented as going into the secondary pipeline (to the `MEM` stage), which is not the observed behavior.
The `MUL` instruction sometimes writes two registers, which causes a delay of one cycle. This was not documented.
Three-operand `PREPARE` instructions take one more execution cycle than stated in the manuals.
The `JMP` and `CTRET` instructions were documented as being one cycle too fast.
A two-operand `MOV` instruction can be combined with a following two-operand arithmetic instructions into a single-cycle three-operand instruction. This behavior was not described in the manuals.
The instruction prefetch mechanism was more effecient than the manuals indicating, speeding up the execution of instructions longer than 32 bits.
Simulator Design
Instructions could continue to flow through the additional pipeline when a long-running instruction was busy in the master pipeline, which was not the case on the real hardware. Only the other way around was possible, with the main pipeline executing instructions while the additional pipeline was executing a long-running instruction.
The `NOT` instruction was not implemented.
Not modeling the read of the special `CTBP` register in the `CALLT` instruction, which missed a data dependence between `CALLT` and other instructions.
The `NOP` instructions was modeled as using the `WB` pipeline stage, which caused extra execution cycles to be inserted in some circumstances.
Simulator Implementation
Signed and unsigned division has a one clock-cycle difference in execution time, and the implementation used the time for the signed case for the unsigned instructions and vice versa.
The bit-manipulation instructions did not keep certain resources locked for their entire execution, letting other instructions slip by, eventually leading to crashes in the simulator.
Hardware Implementation
There is no interlock for successive `SLD` instructions writing the same register, making it possible to destroy data in certain conditions, and leading to faster execution of some scenarios. This condition has been reported as a hardware bug by NEC, and was not modeled in the simulator.

Figure 8.3: Selected bugs found in the V850E simulator

8.2.2 MCore Model

Collins [Col00, Chapter 3][BCF$^+$01] describes the design and short validation of a simulator for a Motorola MCore embedded processor. The target processor is a low-power RISC design with a four-stage pipeline running at up to 100 Mhz. The simulator design very closely follows the processor design, including modeling the data stored in latches between pipeline stages [HP96], which is a higher level of detail than that used in our V850E simulator [Mon00].

The validation of the simulator was performed by running eight small applications on the simulator and comparing the clock cycle counts with a real processor. This tested the basic arithmetic and other instructions. To test the performance and correctness of interrupts and exceptions, a real-time operating system was used on both the simulator and a real processor. The results reported are an error of 100 cycles in one million, or 0.01%, which indicates that for such a simple processor, it is quite possible to construct a cycle-accurate simulator of

Program	Emulator	Initial Simulator			Final Simulator		
			Error			Error	
	Cycles	Cycles	Cycles	%	Cycles	Cycles	%
`fibcall`	325	286	-39	-12.0%	313	-12	-3.7%
`insertsort`	956	1123	167	+17.5%	939	-17	-1.8%
`matmult`	222236	239526	17920	+7.8%	221822	-414	-0.2%
`duff`	1094	1193	99	+9.0%	1083	-11	-1.0%
`fir`	348105	323277	-24828	-7.1%	348095	-10	0.0%
`jfdctint`	4686	5414	728	+15.5%	4733	47	+1.0%

Figure 8.4: Simulator accuracy improvement for test programs

high quality. Collins claims that the remaining error is due to a variable-latency multiplication instruction which is modeled as being fixed-latency.

In general, basing the architecture of the simulator on the processor architecture seems to be a good way to obtain a quality simulator, especially if detailed design information about the hardware is available (or VHDL code).

8.2.3 C167 Model

Atanassov et al. [AKP01] use measurements on hardware to produce a worst-case execution time model for an Infineon C167. They do not attempt to build an exact simulator for the processor, but rather aim for a safe timing model that can be used inside a WCET tool. The result does not model the four-stage pipeline of the C167; instead, it is a set of equations detailing the execution time of an instruction in isolation and considering its interaction with its neighbors, as derived from the hardware measurements (as discussed in Section 5.4.1, the interaction between neighbors are easy to translate into the δ values in our timing model).

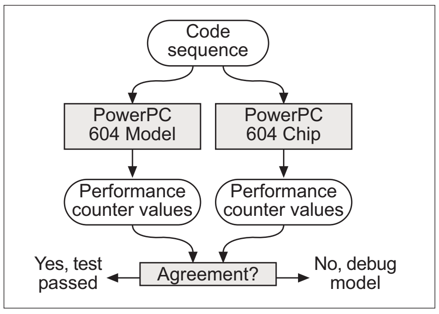
The model was constructed by measuring the execution time for single instructions and sequences of instructions. An external timer triggered via an output pin on the C167 was used for timing measurements, and like Montán [Mon00], instructions were repeated many times to obtain quality measurements. The effect of placing code and data in different types of memory was investigated and incorporated into the model.

The final model was validated by comparing WCET estimates from the analysis tool with the measured WCET results on the target processor, and the results indicate that the model is safe and tight. Most cases show no error, and a few cases give overestimations in the range of 2% to 5%.

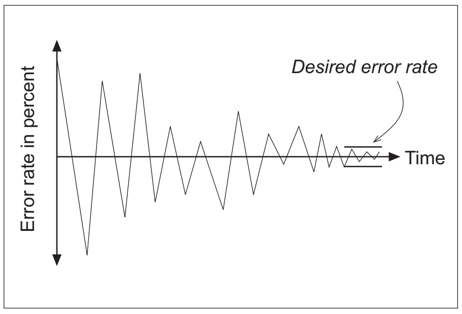
Atanassov et al. note that there it cannot be proven that the model is complete, since it is infeasible to test all possible instruction sequences on the hardware.

8.2.4 PowerPC 604 Model

Black and Shen [BS98] report their experience in validating and fine-tuning a model of a PowerPC processor. The methodology employed is illustrated in



(a) Validation methodology



(b) Development of overall error rate

Figure 8.5: Setup of Black and Shen's Experiments [BS98]

Figure 8.5(a). Many test cases (small programs) are executed on the simulator (PowerPC 604 model) and the real hardware (PowerPC 604 processor). The criterion used for determining whether test cases pass is not clock cycle counts but rather the values of the built-in *performance counters*[2] of the PowerPC, which is only an indirect indication of the execution time. For such a complex processor, however, this is a reasonable approach. As a sanity check, they also execute a number of larger programs and compare the cycle counts on the simulator and the hardware.

The small test cases used are *alpha tests* that execute one instruction at a time to investigate instruction latencies, *beta tests* that check for data dependences between instructions, and *gamma tests* that execute all pairs of instructions to check for inter-instruction interference. *Random* test sequences are employed to test more complex interactions. *Handwritten* test sequences investigate particular properties that are found to be relevant and interesting. The total number of test cases used was about 200000, which was still only a small subset of all the test suites used by the processor designers at IBM and Motorola. Black and Shen strongly recommend the use of as many test suites as possible for timing validation. The small benchmarks all fit in the processor cache, making the results independent of the external memory system, and thus more reproducible and stable.

Black and Shen report that they never manage to get a perfect correspondence between the simulator and the hardware. Fixing an error in the simulator sometimes revealed other errors that had been masked by the first, and the total error could thus increase for some fixes. As illustrated in Figure 8.5(b)[3], over time, the accuracy of the simulator increased (even if some fixes cause the

[2]Most modern desktop processors have an on-chip module that can be used to collect information about how the processor executes instructions, by counting pipeline stalls, cache hits and misses, clock cycles spent, etc. The counters available in such a module are usually called performance counters.

[3]The zero line in Figure 8.5(b) indicates that the simulator and the real processor agrees, while negative values indicate that the simulator underestimates the execution time and positive values indicate an overestimation.

overall error to increase), both for the performance counter agreements and the cycle counts for the large programs. For their final simulator, about 20 percent of the individual small tests failed to be within one clock cycle of the hardware timing, while the large test programs showed an error between 2 and 10 percent.

The errors found were categorized as *specification* (user's manuals, some of which concerning instruction latencies were quite severe), *modeling* (design and implementation of the simulator), and *abstraction* (over-simplifying parts of the architecture in the simulator design).

Their conclusion is that systematic validation is necessary to build a performance model in which users can have confidence, and that achieving a reliable and precise model requires a lot of development effort. The complexity of current high-end microprocessors makes the development of reliable hardware models a "great challenge".

8.2.5 Stanford Flash Models

Gibson et al. investigated the quality of the models used in the development of the Stanford FLASH multiprocessor [GKO$^+$00]. The FLASH project employed a large number of simulators during the development of the system, and thus collected a large amount of data on how simulators and hardware correlate.

The most detailed part of their simulators was the memory controller (since that was at the core of the FLASH research). They also used several different simulators for the MIPS R10000 CPUs used in the FLASH computer.

The testing was initially based on running large multi-processor numerical applications and comparing the resulting execution times. Smaller benchmarks were introduced to help pinpoint specific performance mismatches later in the process.

The mismatches between hardware and simulator were classified as implementation *bugs*, where the behavior deviated from that intended in the simulator design, *omissions* like not modeling the translation look-aside buffers (TLBs) of the real CPU (the same as the abstraction bugs in Black and Shen [BS98], i.e. ignoring relevant hardware details in the simulator), and *lack of detail* where effects are included but not modeled in sufficient detail (simulator design bug).

The CPU models used were all based on generic simulation packages like the SimOS simulators `Mipsy` and `MXS` [Her98], which were given parameters to closely emulate the R10000. However, these models were unable to capture all relevant details of the CPU, giving very different execution time results for key parts of the code like TLB miss handlers. This is claimed to be a general problem with generic CPU simulators: they are not sufficiently configurable to model all the corner cases of the real hardware. Certain simplifying assumptions have to be made.

Gibson et al. conclude that that simulation technology is barely able to keep up with the increased complexity of modern computer systems, and that it is necessary to compare the simulation results to hardware in order to build a good model. By calibrating models to match the hardware as it is being made available, good models can be obtained, but without validation against the actual hardware, simulation results should be treated with caution.

8.2.6 Alpha 21264 Model

Desikan et al. [DBK01] validated a simulator for the superscalar Compaq Alpha 21264 processor [Com99], built from the SimpleScalar toolkit [ALE02] with added code to model the intricacies of the Alpha 21264 pipeline. The validation was performed by comparing the simulator with a Compaq workstation based on the processor. A small number of microbenchmarks were used in this validation effort, and the average number of instructions per cycle (IPC) was used as the main comparative metric. The measurements on the hardware was accomplished by using profiling tools that accessed the processor's performance counters.

The microbenchmarks attempted to exercise only parts of the processor pipeline, by keeping the load on other parts of the pipeline low. The error rate on the microbenchmarks was brought down to around 2%, with one outlier at 11%. Desikan et al. list several known discrepancies in their model that could be the cause of the remaining error. It should be noted that the microbenchmarks all fit within the cache of the 21264, making them quite independent of the behavior of the external memory system.

Extending the experiments to larger programs from the Spec suite, the error rate varied between -39% and +40% (still counting IPC). The main reason for the residual error was not in the processor core modeling but in the memory system, especially regarding the handling of page faults and DRAM access latencies. The authors point out that this is very hard to simulate in a precise manner, due to the many and complex interactions in the memory system.

They compared their validated simulator with a simplified simulator that only models a few of the performance-enhancing features of the Alpha 21264, and the generic `sim-outorder` simulator from the SimpleScalar toolkit, with Alpha 21264-like parameters. The simplified simulator underestimates performance, since it only models the limitations of the processor, while the generic simulator overestimates the performance since it doesn't model many of the limitations present in real processors. Their conclusion is that since simulators (and real machines) can have different bottlenecks limiting the performance, simulation of new features intended to enhance performance can come to very different conclusions depending on the simulation model used, which is a problem for computer architecture research.

From a WCET analysis standpoint, this work points to the importance of validating simulators against real hardware, and provide further evidence that generic simulators typically do not provide the level of detail required to obtain reliable timing models. Furthermore, the large number of details not modeled in the validated simulator due to missing information points to the need to use hardware that is simple and well-documented if precise simulators are desired.

8.3 Other Hardware Aspects

Apart from processor pipelines, there are many other hardware features that must be modeled correctly to enable a safe WCET analysis to be performed. This section will give a quick overview of some of the problems.

8.3.1 Caches

The first type of cache to be analyzed in WCET analysis was direct-mapped caches [LBJ$^+$95, HAM$^+$99], since their behavior is easy to model and predict. Direct-mapped caches are popular in real systems thanks to their simple implementation, and there is no need to cut corners or make simplifications to the implementation. Thus, direct-mapped caches can be considered to correspond perfectly to the models (modulo hardware implementation bugs).

For set-associative caches, things get more complicated. Most analyses for set-associative caches in the literature [LBJ$^+$95, LMW96, OS97, Sta97, FMW97, Mue00] assume that the replacement condition is perfect LRU, Least-Recently Used [HP96].

LRU is simple, understandable, gives good performance in real life, and is easy to model, but it is not generally practically implementable in hardware when associativity is greater than four, due the effort required to track the latest access date for all cache lines in a set. For example, a two-way associative cache only requires one bit of information per cache set (which was most recently used), while at four ways associativity, you need six bits of state per set of cache lines for a practical implementation [Mot90]. For a 32-way set associative cache, the minimal number of bits per set is 118, while a practical implementation uses 160 bits [Han98]. In addition to the memory overhead in the cache, there is also the question of the logic required to update the LRU information, and its impact on the number of read and write ports required to the cache array. The logic is burdensome enough that Intel's four-way caches are only pseudo-associative, in order to increase the clock speed of the CPU [Han98].

For cost reasons, some set-associative cache systems for embedded systems use other replacement policies. The Intel XScale uses a *round-robin* cache replacement scheme for its 32-way set-associative cache [Int00a] and the ARM 710T, 720T, and 740T macrocells all use a 4-way unified cache with *random* replacement policy [ARM98b, ARM98c, ARM00a].

One way of handling (pseudo-) random cache replacement and round-robin replacement is to treat the cache as a direct-mapped cache [FHL$^+$01]. This is safe but quite pessimistic. The pessimism depends on the size of the sets (the Coldfire 5307 analyzed in [FHL$^+$01] is four-way set associative), and could be very bad for high-associativity caches like those of the XScale. It is not feasible to track the pseudo-random sequence generated by the random number generator on a processor, since this depends on knowing the precise number of cache misses since system start. Furthermore, the random sequences can be rather long. The ARM7 series use a 16-bit linear feedback register, providing a random-number sequence that contains 32768 distinct values before starting over [ARM98a].

This points to a need for research into predictable cache architectures, and an awareness of the great variety of caching systems in use for embedded systems. Just like the models of processor pipelines, cache models need to be validated against the actual caches used.

8.3.2 Memory-Management Units

For a system using a memory-management unit (MMU), there is a need to analyze the worst-case timing of memory accesses in more detail. The time required to load a value from memory can be quite high (even for fast memory and no cache), since a translation-lookaside buffer (TLB) miss can require a complex table walk in main memory, involving several memory accesses. Some techniques to reduce this penalty are presented in [BA01].

8.3.3 Dynamic Random-Access Memories

Dynamic RAM (DRAM), unlike static RAM (SRAM), has to be periodically refreshed to retain its data. This refresh will occasionally make the memory unavailable, leading to unpredictable delays to a program. The effect can be measured to be a few percent of execution time, which has to be accounted for, and no practical analysis has yet been presented to address the problem [AP01].

A related issue that (as far as we know) has not received any research attention is WCET analysis of the complex timing behavior of modern memory subsystems like RDRAM and SDRAM, where memory latencies depends on the previous accesses.

8.4 How to Build a Good Simulator

From the information presented in this chapter, we can draw some conclusions on how to build quality processor simulators for WCET analysis can be drawn.

8.4.1 Do Not Trust Manuals

Hardware manuals cannot be considered a reliable source of information, and other sources of information are necessary to construct good models. They are useful as a starting point for a simulator design effort, but should always be validated by experiments on the hardware.

8.4.2 Validate Against Hardware

It is absolutely necessary to validate a simulator or other model of any hardware against the real hardware. In addition to handwritten code sequences designed to test known difficulties, the validation must include testing using real programs and random code sequences in order to make sure that as many bugs as possible from the simulator design phase are caught.

If a VHDL RTL model of a processor is available, it can be used to validate the software model by running tests of the two against each other. If this can be automated, such tests can be run for weeks or months, reducing the number of errors in the hardware model successively.

8.4.3 Use Simple Processors

Contrasting the experience with the V850E, MCore, and C167 with that from the PowerPC, Alpha, and Stanford FLASH, the conclusion is that it is much easier to build models for simple processors. The simulators for the simple processors show very good agreement between simulators and hardware, while the results for the complex processors are less precise.

The border between "modelable" and "not modelable" is not clear-cut. As complexity is added, models get increasingly complex to build, but we believe that processors with a complexity like the V850E are modelable. Dynamic superscalars could be manageable, but out-of-order processors are definitely too complex to model with current techniques.

Hughes et al. [HPRA02] note that even if absolute agreement with the hardware is not achievable, a model can still be useful for architectural research as long as it allows good predictions of the relative impact of various features. For WCET analysis, however, we require absolute agreement with the hardware.

8.4.4 Avoid Generic Modeling Packages

Generic processor modeling tools like SimpleScalar are good for experimenting with certain processor design ideas, but they are in general not able to capture all the details of a particular real-life processor. Speed of implementation and convenience is bought by simplifying many factors and providing general approximations, making it very hard to model the peculiar design choices of a particular processor.

Chapter 9

Prototype and Experiments

We have implemented a prototype tool incorporating the pipeline timing analysis and timing model described in this thesis. The tool does not yet use automatic flow analysis and global low-level analysis has not been needed for the targets and programs we have studied. Thus, the current tool is a subset of the full architecture in Figure 3.1, as shown in Figure 9.1.

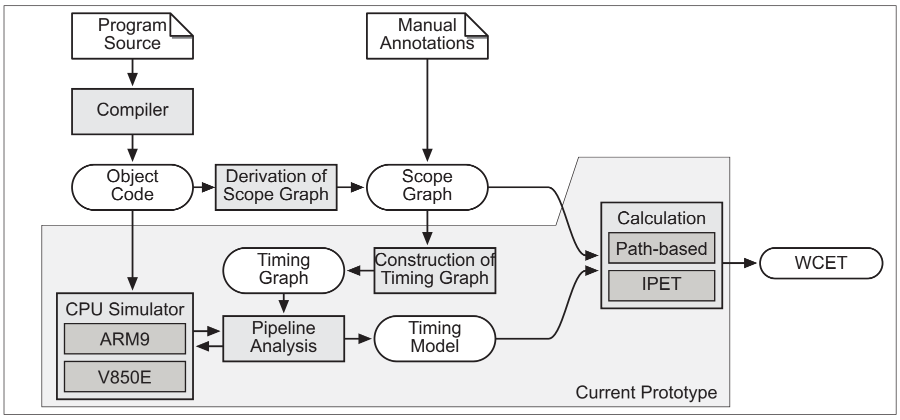


Figure 9.1: WCET tool prototype implementation

The tool is currently a command-line tool that runs under Solaris, Linux, and Windows 2000. We have implemented two different CPU models, V850E and ARM9 (as well as two variants of the V850E with slower memory timing). It is possible to select the CPU model by using a `--cpu` option on the command line for the tool. There are also two calculation modules, one IPET-based [EE99] and one path-based [SEE01b], which can be selected independently of the CPU model used using a `--mode` option. This independence shows that it is quite easy to port the tool to new targets and exchange components, thanks to our architecture.

For ARM9 programs, we parse the object code in ELF format directly and construct a scope graph from it, while V850E programs rely on a modified IAR V850/V850E C/Embedded C++ compiler [IAR99] (we have only tried C programs) to generate object code files on a special format which are then used to construct the scope graph.

We have implemented a cache analysis in the style of Ferdinand et al. [FMW97], just to check that it was possible to deposit information about cache hits and misses in the scope graph. However, we are not using it in our experiments, since none of our targets currently use a cache.

Currently, the tool only generates WCET estimates. An industrial tool has to generate best-case estimates as well as a worst-case estimates, since this allows a user to get a rough idea of the tightness of the WCET estimate. Implementing BCET analysis in addition to WCET analysis should not be too difficult, and has been done by several other WCET research groups [HAM+99, ZWR+01].

9.1 Test Programs

Program	**Description**	**Properties**
`compress`	Compression using lzw.	Nested loops, goto-loop, function calls.
`crc`	Cyclic redundancy check computation on 40 bytes of data.	Complex loops, many decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.
`duff`	Using "Duff's device" [Ray00] to copy a 43 byte array.	Unstructured loop with known bound, `switch` statement.
`expint`	Series expansion for computing an exponential integral function	Inner loop that only runs once, structural WCET estimate gives large overestimate.
`fibcall`	Simple iterative Fibonacci calculation, used to calculate fib(30).	Parameter-dependent function, single-nested loop.
`fir`	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	Inner loop with varying number of iterations, loop-iteration dependent decisions.
`insertsort`	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $n^2/2$ iterations.
`jfdctint`	Discrete-cosine transformation on a 8x8 pixel block.	Large basic blocks, single-nested loops.
`lcdnum`	Read ten values, output half to LCD	Loop with iteration-dependent flow.
`matmult`	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loop.
`ns`	Search in a multi-dimensional array	Return from the middle of a loop nest, deep loop nesting.
`nsichneu`	Simulate an extended Petri Net	Automatically generated code containing massive amounts of `if`-statements (> 250)

Figure 9.2: The test programs used in our experiments

In our experiments, we have used a number of diverse test programs, as

	No Pipeline		**Pipeline**		**Actual**	**No pipe**
Program	**cycles**	**+%**	**cycles**	**+%**	**cycles**	**-pipe**
`compress`	10388	19.9%	8672	0.12%	8662	19.8%
`crc`	61340	102.6%	30271	0.0%	30271	102.6%
`expint`	10062	17.2%	8588	0.0%	8588	17.2%
`fibcall`	559	78.6%	313	0.0%	313	78.6%
`fir`	487808	40.1%	352073	1.14%	348095	39.0%
`insertsort`	2328	116.8%	1794	67.0%	1074	49.7%
`jfdctint`	5388	9.4%	4942	0.35%	4925	9.1%
`lcdnum`	341	72.2%	198	0.0%	198	72.2%
`matmult`	275859	24.4%	221824	0.0%	221824	24.4%
`ns`	20653	48.3%	13934	0.04%	13928	48.2%
`nsichneu`	87193	70.6%	51133	0.03%	51116	70.5%

Figure 9.3: Execution time estimates with and without pipeline effects

presented in Figure 9.2. The goal has been to collect programs with many different types of flows and structures, to thoroughly test all aspects of the low-level analysis and the calculation. Most of the programs have been used by other WCET research groups[1], and some have been specially crafted to test particular aspects of calculation.

Actual execution time values for these programs were obtained by running a worst-case trace of the programs through the same simulator used for the WCET analysis. Thus, these experiments only deal with the effectivenes of pipeline modeling and do not take into account any differences between the simulator and the actual hardware. The trace was obtained by manual analysis of the programs, and extensive testing was performed to make sure that they really correspond to the worst case.

9.2 Usefulness of Timing Effects

To demonstrate the value of modeling the pipeline overlap between basic blocks (pairwise timing effects), we generated WCET estimates for our test programs both with and without timing effects. By ignoring the timing effects, we do not model the pipeline overlap between basic blocks. Figure 9.3 shows the results of the experiments, using the V850E processor model and path-based calculation.

The columns under “No pipeline” shows the WCET estimates achieved assuming all timing effects are zero, while “Pipeline” shows the estimates achieved when timing effects were used. The “+%” columns shows the overestimate compared to the execution time in the “Actual cycles” column. The last column, “No pipe-pipe” shows the difference in overestimation between the “No pipeline” and

[1]Despite this fact, comparing experimental results is very difficult since all groups target different processors and use different calculation methods.

Program	**IPET with flow** cycles	+ %	**Actual** cycles	**Facts** used
`crc`	30271	0.0%	30271	6
`duff`	1083	0.0%	1083	1
`expint`	8588	0.0%	8588	4
`fibcall`	313	0.0%	313	0
`insertsort`	1074	0.0%	1074	1
`lcdnum`	198	0.0%	198	2
`matmult`	221824	0.0%	221824	0

Figure 9.4: Estimated and actual execution times, with known program flow

"Pipeline" numbers, and is a measure of the tightness gained from the pipeline modeling by timing effects.

The flow facts that the path-based calculation can use were applied to generate the best possible estimates, but some programs still exhibit execution time overestimations due to program flow, especially `insertsort`.

In general, modeling pipelines seem to tighten WCET estimates by at least 20%, with some programs gaining much more. The benefit is greatest for programs with many small basic blocks (like `crc`, `fibcall`, and `nsichneu`), and least for programs with large basic blocks (like `jfdctint`). Experience from compiler design and computer architecture indicates that the case with many small basic blocks is quite common [HP96, Muc97], and thus modeling effects across basic blocks is essential for a tight WCET analysis. In an analysis of overestimation sources in WCET analysis, Kim et al. also reached the conclusion that pipeline effects between basic blocks is a very important factor [KHM99].

Note that the measurements in Figure 9.3 do not include `duff`, since it is an unstructured program and cannot be handled by the path-based calculation.

9.3 Precision

The precision of the pipeline analysis can be evaluated by performing WCET analysis for programs with known worst-case flows. WCET estimates are generated using flow information that perfectly models the flow of the program, and thus any overestimate observed will have been caused by the timing model and not the flow information. We have such precise flow information for some of our test programs, requiring the use of the IPET-based calculation module to handle all the facts.

Note that to eliminate the possibility that errors in the calculation module mask errors in the pipeline model, the program execution profiles (execution counts for basic blocks in the program) generated by the calculation were checked against the execution counts for the basic blocks in the worst-case traces [EES01b].

Figure 9.4 shows the results of running the programs where the flow is perfectly modeled, using the V850E pipeline model. The columns under "IPET with flow" shows the WCET estimate from the tool. The "Actual cycles" column shows the actual WCET for the program. The column "Facts used" shows the number of facts needed to achieve the precision (each fact states an execution count bound for a particular block, or an infeasible path). We can see that the execution times agree perfectly for all the programs, which indicates that no precision is lost in the pipeline model, if used with a powerful enough calculation module. Usually, not too many flow facts are needed to achieve good precision in the WCET analysis. Here, `duff` is included, since our IPET-based calculation module can handle unstructured programs.

Compared to other work in WCET analysis, the precision we achieve is competitive. Stappert and Altenbernd report overestimations between 4% and 13% for straight-line code on a superscalar PowerPC 604 processor with caches [SA00]. Healy et al. reports 0% overestimation (perfect agreement) for predictable programs like `matmul`, and up to 100% overestimation for a sort program similar to `insertsort` (due to the limited power of the calculation method used), considering only the pipeline of the MicroSPARC 1 [HAM$^+$99]. Ernst and Ye report overestimates ranging from 0% to 10%, analyzing a SPARC pipeline similar in complexity to our V850E, using IPET calculation [EY97]. Colin and Puaut report overestimates between 1% and 265% for a simplified integer-only Pentium CPU with caches and branch prediction [CP00]. Lundqvist and Stenström report overestimates between 0% for programs with predictable flow, up to 1600% for programs with very irregular flow [LS99a].

Note that not all published work include measurements comparing the generated WCET estimates with actual execution times; in many cases, the effectiveness of analysis methods are evaluated by comparing the estimated and actual number of cache misses etc., and not actual execution times.

9.4 Computation Time

In order to determine the computational costs of the pipeline analysis, we measured the time required for the pipeline analysis part of our WCET tool prototype for each of the test programs. Note that we exclude the time required to read the input files and perform the calculation, since they are not really related to the analysis.

Figure 9.5 shows the time required to perform the pipeline analysis for our test programs. "Pipe time" shows the time required for the timing analysis. In order to give an appreciation for the magnitude of the time, the column "Load time" shows the time taken to read the input files into the tool, and "Sim time" the time required to run the trace corresponding to the worst-case execution of the program in the simulator. The column "Pipe/Sim" shows the relation between the time for the pipeline analysis and simulation of the worst-case execution. All times are in seconds, obtained on a PC with a 700 Mhz Pentium III processor, 768 MB of RAM, and Windows 2000 SP2. Our tool was compiled in release mode with Visual C++ 5.0.

Program	Load Time	Pipe Time	Sim Time	Pipe/Sim
`compress`	0.24	0.52	0.33	1.58
`crc`	0.10	0.19	1.58	0.12
`duff`	0.03	0.09	0.07	1.31
`expint`	0.05	0.13	0.19	0. 68
`fibcall`	0.01	0.03	0.02	1.50
`fir`	0.04	0.08	17.55	$\approx 0\%$
`insertsort`	0.03	0.04	0.07	0.57
`jfdctint`	0.17	0.23	0.20	1.16
`lcdnum`	0.03	0.12	0.01	12.00
`matmult`	0.08	0.18	11.29	0.02
`ns`	0.03	0.09	0.76	0.12
`nsichneu`	3.60	5.20	2.98	1.74

Figure 9.5: Computation times (in seconds of wall clock time)

The time taken for the pipeline analysis is roughly linear to the size of the program, as measured by the time taken to load the code. Typically, it takes about twice as long as loading the program code (which is linear to the size of the program).

Compared to the time required to simulate the worst-case execution, the pipeline analysis is generally efficient. For programs that are loop-intensive, the pipeline analysis time ("Pipe time") can be much smaller than the time to run the program on the simulator (`fir` and `matmult` are the best examples). The numbers for `lcdnum` shows that there are cases when the analysis can take longer, which is explained by the high branching factor of a `switch` statement in the `lcdnum` code (in Figure 9.7, we can see another effect of this in that `lcdnum` requires investigating quite many long sequences).

9.5 Long Timing Effects

As discussed in Section 7.1.3, by extending the memory latency in our V850E model from one to six cycles, we provoked long pipeline effects. This is an artificial test not corresponding to any particular real setup of the V850E, but it is a useful way to test the extension condition and timing analysis method.

Figure 9.6 shows the number of times and timing effects found; to make the table easier to read, zeroes have been replaced by dashes. Sequences of length one correspond to nodes in the timing graph, and length two to the simple timing effects between neighboring nodes. Sequences of length three and up are long timing effects. Note that we are discussing *nodes* in the timing graph, and that each node can contain several instructions.

Note that all long timing effects found were negative, even though the V850E has the potential to exhibit positive long timing effects, as discussed in Section 7.1.3.

	Length of timing effects						
Program	**1**	**2**	**3**	**4**	**5**	**6**	**7**
`compress`	106	142	19	7	-	-	-
`crc`	45	58	4	-	-	-	-
`duff`	18	27	2	-	-	-	-
`expint`	24	30	-	-	-	-	-
`fibcall`	7	8	-	-	-	-	-
`fir`	14	18	-	-	-	-	-
`insertsort`	7	8	2	1	-	-	-
`jfdctint`	14	13	6	3	-	-	-
`lcdnum`	26	43	3	2	-	-	-
`matmult`	36	43	10	3	-	-	-
`ns`	18	22	1	-	-	-	-
`nsichneu`	754	1377	2	1	-	-	-

Figure 9.6: Times, timing effects, and long timing effects found

The frequency of long timing effects vary significantly across the test programs. We note that no programs feature long timing effects longer than four nodes. The frequency of occurence of long timing effects compared to the number of effects across two nodes varies from very rare (`nsichneu`), to very common (`matmult`).

We validated that we have found all timing effects by using a much more conservative extension condition with the same set of programs, finding the same set of timing effects.

	Length of sequences analyzed						
Program	**1**	**2**	**3**	**4**	**5**	**6**	**7**
`compress`	106	147	120	48	13	1	-
`crc`	45	58	44	15	-	-	-
`duff`	18	27	32	7	-	-	-
`expint`	24	30	29	11	-	-	-
`fibcall`	7	8	9	5	-	-	-
`fir`	14	18	17	11	6	1	-
`insertsort`	7	8	6	4	1	-	-
`jfdctint`	14	16	16	8	-	-	-
`lcdnum`	26	43	59	23	2	1	-
`matmult`	36	43	49	28	12	6	-
`ns`	18	22	28	18	2	-	-
`nsichneu`	754	1378	1009	2	2	-	-

Figure 9.7: Length of sequences investigated

To determine the run-time cost of the `extcond-scalar` extension condition used for the V850E visavi the timing effects found, we counted the number of sequences of a given length that were analyzed for each benchmark, with the results shown in Figure 9.7. The number of sequences examined goes down quickly after length three, since fewer and fewer sequences will need to be extended according to the extension condition.

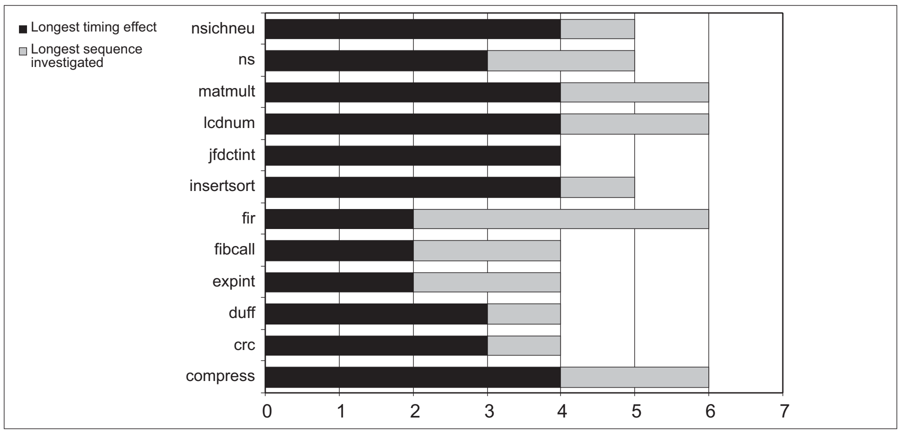


Figure 9.8: Longest timing effects vs. longest sequences investigated

Comparing the lengths of sequences investigated shown in Figure 9.7 with the length of the timing effects shown in Figure 9.6, we see that we typically examine a few sequences that are longer than the longest timing effect found, but not very many.

Figure 9.8 shows the length of the longest timing effect found vs. the length of the longest sequence examined for the test programs. We note that for `jfdctint`, the maximal timing effect and sequence length are both four, which is a very good result, while `fir` is worse, with a maximal timing effect length of two but requiring sequences up to length six to be examined. In general, examining sequences two nodes longer than the length of the longest timing effect found seems to be typical. This indicates that a more intrusive extension condition could conceivably do better.

Program	**Load time**	**Pipe time**	**Sim time**	**Pipe/Sim**	**LTE/noLTE**
`compress`	0.24	0.70	0.44	1.59	1.35
`crc`	0.10	0.22	1.63	0.14	1.16
`duff`	0.03	0.11	0.08	1.47	1.29
`expint`	0.05	0.13	0.20	0.65	1.00
`fibcall`	0.01	0.03	0.21	0.14	1.00
`fir`	0.04	0.10	21.59	0.00	1.25
`insertsort`	0.03	0.05	0.08	0.63	1.25
`jfdctint`	0.17	0.25	0.22	1.14	1.08
`lcdnum`	0.03	0.13	0.01	13.00	1.08
`matmult`	0.08	0.24	12.48	0.02	1.33
`ns`	0.03	0.10	0.83	0.12	1.11
`nsichneu`	3.10	5.77	4.22	1.37	1.11

Figure 9.9: Computation times for pipeline analysis with long timing effects

Figure 9.9 shows the execution time of the timing analysis when long timing effects are present. The columns "Load time", "Pipe time", "Sim time", and "Pipe/sim" have the same meaning as in Figure 9.5. The last column, "LTE/noLTE" shows how the timing analysis time when long timing effects are present relates to the time when no long timing effects are present. This number shows that the analysis gets a little more expensive, but not inordinately so. The values in the column "Pipe/sim" are very similar to Figure 9.5, also demonstrating that long timing effects do not cause any explosion in the complexity of the analysis.

9.5.1 Long Timing Effects and Precision

	IPET, no LTE		Actual
Program	**Cycles**	**+%**	**Cycles**
`crc`	34427	1.5%	33914
`duff`	1705	16.2%	1467
`expint`	8720	0.0%	8720
`fibcall`	332	0.0%	332
`insertsort`	2361	0.2%	2356
`lcdnum`	268	12.6%	238
`matmult`	312550	9.5%	285384

Figure 9.10: Effect of long timing effect on precision of WCET estimates

Considering the effect on the precision of the execution time estimates, Figure 9.10 shows the results of doing WCET analysis with the slow data memory on the V850E. The "IPET, no LTE" columns shows the execution time estimates generated by ignoring all timing effects for sequences of length three or longer. We only use the programs where we have perfect flow information, since we do not want uncertainty about the program flow to disturb the experiments. We can see that for the programs that do not exhibit long timing effects (`expint` and `fibcall`), the result is still a perfect estimate. For the other programs, we get an overestimate since the long timing effects observed are all negative, and ignoring them thus increases the execution time estimate[2].

The numbers indicate that for some programs, the consequences of long timing effects can be quite substantial, while it hardly matters for other programs. In general, we claim that long timing effects must be modeled in order to generate safe and tight WCET estimates. As always, the effect varies with the processor architecture and the properties of the program code.

[2]We do not show numbers for the timing estimates generated by calculating WCET with the long timing effects, since they would be identical to the actual execution times. The interesting question that we are trying to answer is how much precision is lost by *ignoring* the long timing effects.

Chapter 10

Conclusions and Discussion

Static WCET analysis is about overcoming the obstacle of execution time variability in determining the WCET of a program. Compared to measurements, static WCET analysis offers the ability to obtain guaranteed safe estimates, with no risk of missing the worst case due to a bad choice of test cases or bad luck with hardware variability. In this thesis, I have presented five contributions to the state of the art in worst-case execution time (WCET) analysis.

In the field of fundamental theory, a *formal mathematical model* (Chapter 5) has been used to reason about and investigate the safety and tightness of WCET analysis. Considering the problem of building correct WCET analysis methods, a discussion on how to build *hardware models* was given in Chapter 8. On the practical technique side, I have presented a *low-level timing modeling scheme* (Chapter 4) and a corresponding *timing analysis method* (Chapter 6) to generate the timing model for a program. The overall *tool architecture* presented in Chapter 3 shows how the timing model and timing analysis are integrated with other parts of the WCET analysis problem.

Overall, these contributions aim towards building retargetable, flexible, efficient, broad applicable, and correct static WCET analysis tools for real-time embedded systems. Considering each of these properties:

Retargetability is supported by the timing model and timing analysis since they isolate the model of the target processor from the other parts of the tool (as specified by the tool architecture). The architecture allows other analysis modules to be chosen to suit the properties of a particular target processor, allowing the efficient reuse of existing modules.

The timing analysis is designed to allow the use of any trace-driven simulator for the target processor, which makes it possible to use already existing simulators, reducing the time required for adapting the analysis to new targets. There is no need to construct a special-purpose timing model just for WCET analysis, as is the common practice in WCET research. Proven and familiar simulators can thus be used a new way, making WCET analysis easier to sell to engineers. We have demonstrated the retargetability by implementing our analysis for two targets, the NEC V850E and the ARM9.

Flexibility is enhanced by the tool architecture since it is easy to add new analyses to a modular tool. The set of components active for any particular analysis instance can easily be customized. The interface data structures allows the results of several analyses to be combined. Compared to other WCET tool architectures [FKL$^+$00, CP01a], the interface data structures only assume that a program is structured as a set of basic blocks with flows between them, but no restrictions are made considering the flows. The architecture is also flexible enough that if needed, it would be possible to implement an integrated cache and pipeline analysis method within our framework, even if it would not be true to its spirit.

In our prototype, we have implemented two calculation modules, one path-based and one using IPET. The modules can be chosen independently of the target processor, thus demonstrating that modules can be replaced independently of other modules.

Efficiency is achieved by separating WCET analysis into modules, each of which performs a single well-defined task. Trade-offs between speed and precision can be performed by using different versions of a component, which makes it possible to select a good compromise for a particular program and target hardware.

The timing analysis method presented in this thesis is efficient, typically running in time linear to the size of the program. The timing model has been used to implement some very efficient calculation methods [SEE01b]. Using a prototype tool, we have evaluated the speed of the timing analysis, and the results indicate that we quickly generate precise timing models.

Broad applicability has been designed into the timing analysis method and timing model. The timing model is independent of the particulars of the processor architecture, supports any program structure (even spaghetti code, as demonstrated by including an unstructured test program in our experiments), and can represent the timing of a wide spectrum of embedded processors (the model might have to be approximate in some circumstances, though). The timing analysis is likewise hardware independent, capable of handling any program structure, and applicable to a wide spectrum of embedded processors.

Verifying the *correctness* of a tool is facilitated by the tool architecture since each component can be tested and validated in isolation. Errors in one component cannot compensate and mask errors in another component, as is the case for validation dealing with an entire tool as a black box. Since each component is comparatively simple, its implementation is simpler and therefore less error-prone.

Understanding the issues involved in modeling hardware is crucial to building correct WCET analysis tools, and to build systems that are amenable to static WCET analysis. The discussion on hardware models gives some concrete advice on how to build correct processor simulators and how to validate simulators.

The formal pipeline model (Chapter 5) allows us to show under which circumstances our timing analysis method is safe, and explores the theoretical limits of precise pipeline timing modeling and analysis. It has been shown that for some classes of pipelines, WCET analysis will always be safe since only negative timing effects can occur. There are other classes of pipelines where positive timing effects can occur and even propagate arbitrarily, requiring conservative models to achieve a safe analysis. For future research in WCET analysis, the model provides a formal framework for investigating pipeline timing modeling and analysis.

10.1 Scalability of Our Approach

Most of the test programs presented in Chapter 9 are quite small, and do not really stress our prototype tool and computers. Hence, it is natural to question the scalability of the proposed pipeline analysis method. However, the results for `nsichneu` are encouraging: even for a program 50 times bigger than most of the other test programs, the time for the analysis is proportional to the size of the program. Furthermore, theoretical arguments can be made for the almost-linearity of the analysis: since each basic block in the program is only visited a few times in the analysis, the analysis should be roughly linear in the program size.

We should note that the experiments show that the precision of WCET analysis and the computation time required to perform the analysis varies greatly between test programs. The structure of flow of a program, the quality of the object code generated by a compiler, the size of basic blocks, and the processor used: all these factors affect the time taken for the pipeline analysis[1]. The largest complexity risk in this regard probably comes from the processor architecture: if very long timing effects are common, the timing analysis will have to examine many long sequences, and the computation time will increase.

The main performance bottleneck for scaling WCET analysis as a whole to bigger programs is probably the calculation. We have demonstrated that, by using our timing model, path-based calculations can be performed in almost linear time, even when taking certain flow facts into account [SEE01b]. However, for maximal precision and flow expressiveness, IPET calculation seems necessary. Thus, one possible solution is to use different calculation algorithms for different parts of a program, depending on the complexity of the flow facts for that part of the program.

10.2 Practical Capability of WCET Analysis

As discussed in Chapter 2, a lot of research effort has been spent on static WCET analysis. The result of this is a body of knowledge that should allow us

[1]This is not surprising; all research in static program analysis suffers from the same fact of life: the quality of the results is dependent on the programs an analysis method is applied to. Unless the application domain is very narrow, there is no such thing as a "typical" program that can be considered representative for all programs.

to build usable WCET analysis tools. In this section, we will try to summarize the current state of WCET research, in order to ascertain the maturity of WCET analysis techniques and the practicality of implemeting WCET analysis tools.

10.2.1 Flow Analysis

The complexity of the flow of a program seems to be the greatest determiner for the WCET estimation precision achievable for a particular program. Most WCET research groups report results that indicate a tight analysis of the hardware, while the program flow sometimes causes large overestimations.

Applying WCET analysis to real-life systems requires assistance in determining the program flow. A practical tool has to be pretty much one-click analysis, where the initial analysis can then be sharpened by adding user information. Of course, there are cases when user intervention is needed to get a result at all, like complex loops that cannot be automatically bounded. It is also important that the results of the flow analysis are presented to the user to enhance the understanding of the program.

Automatic flow analysis methods are currently limited to well-written programs with well-structured loops. In most cases, only counted loops can be automatically bounded, while loops that use shift operations or other non-arithmetic operations to manipulate the loop index have to be manually bounded. Also, using function pointers or unstructured code breaks most analysis methods. However, even with these limitations, flow analysis methods are good enough to provide decent assistance for real-time programmers, with manual intervention necessary in tricky cases. The useability is not as good as compilers, and could maybe be compared to `lint`-style tools, where the users have to invest some effort in configuring the tool before obtaining useful results.

Since flow analysis can fall back to the great body of research in program analysis and program understanding, it is relatively easy to show that a certain analysis method is correct.

The information used to guide the flow analysis should be represented in a robust format that lets the information survive changes to surrounding code [HLS00b]. Our flow facts language is excellent for flow information to be used in calculation, but is invalidated by recompiles since it depends on the structure of the object code. Thus, flow facts as they are currently designed are not suitable for source-code annotation.

10.2.2 Global Low-Level Analysis

On the hardware side, pipelines and caches have to be modeled if the target systems includes such features, since otherwise the estimates could potentially be off by several hundred percent [KHM99].

Since the access pattern of the instruction cache can be determined from the program flow graph (at least when speculatively execution is not being used), instruction cache analysis is quite easy. Current techniques generate good estimates for the caching behavior, at least for single-level caches. However, some cache designs (like using random replacement policies) makes it hard to obtain tight results. Reported results for direct-mapped and set-associative instruction caches indicate that very high precision could be achieved in practice.

Data caches are harder to analyze tightly, since the addresses accessed from a certain memory access instruction can vary across the execution of a program. No really satisfactory solution to this problem has been presented, only partial solutions that decrease the pessimism for certain classes of accesses.

Unified caches pose a much harder problem since they mix the predictable accesses to the program instructions with the unpredictable accesses to data. They can be analyzed at a great cost in computational complexity, but should be avoided since they decrease the precision achievable.

Similarly, most hardware tricks aimed at improving the performance of caches reduce the modelability and thus predictability. Examples are critical-word-first cache refills, non-blocking caches, and victim caches.

Considering branch prediction, practical analyses have been presented for simple predictors. For complex multilevel predictors, the complexity of the analysis is daunting, and it is not clear that there is a practical method to analyze them.

Demonstrating the correctness of global low-level analyses should not be overly difficult, since they work in rather simple domains. However, it is necessary to show that the hardware used actually implements the architecture assumed in the analyses.

10.2.3 Local Low-Level Analysis

For simple processors without pipelines, local low-level WCET analysis is trivial. Just assign an execution time to each instruction, without regard to the neighboring instructions.

Pipelines makes things more complicated, but the analysis for pipelines can still be made very precise, at least for simple pipelines. Almost all research has targeted processors with in-order issue and at most two parallel pipelines, and for these processors, the results indicate that very tight timing analysis is possible.

No practical analysis for out-of-order superscalar processors has been presented. Due to the problems of very complex behavior, timing anomalies, and the problem of finding all relevant processor features, we do not think that a practical, safe, and tight analysis is feasible[2].

Proving the correctness of pipeline analysis is much harder than for flow analysis methods or global low-level analysis, since pipeline analysis needs to

[2]It is possible that using a timing model like ours with some fixed maximum length of sequences to investigate would give a reasonable approximation of the execution time of an out-of-order superscalar processor, since the effects of the scheduler would tend to even out across longer streches of code. However, there would be no guarantees about the safety of the analysis, but the results would probably be rather good on average.

model some very complex hardware. The best we can do is to validate the hardware models used against the hardware, which is time-consuming. It is also very hard to show that all cases that can occur on the hardware have been found and accounted for in the model. Thus, in the best case we have an analysis which is highly likely to be correct.

Considering the impact of the memory system (ignoring the caches), different memory areas with different access speeds can be easily handled. Delays due to DRAM refresh are hard to analyze, and no really satisfactory solutions have been presented.

10.2.4 Calculation

The calculation methods presented in the literature gives us a number of choices in trading flow precision for speed of computation.

Tree-based approaches are very fast, but have problems taking advantage of flow information and handling programs with irregular flows.

IPET is the most powerful technique presented, and seems capable of handling almost any kind of program flow, including unstructured programs. Many different kinds of flows can be expressed. In most cases, the calculation is efficient, even if there is a theoretical potential for a complexity explosion.

Path-based approaches take a middle ground, handling more complex flows than tree-based approaches, but less than IPET. They are generally efficient, but if complex flows are modeled, many paths may need to be examined to find the longest path, making the approach less efficient.

Correctness is quite easy to show for calculation methods, since they operate in a simple and well-defined domain (assuming that we do not integrate the low-level analysis with the calculation).

10.3 Building Analyzable Systems

In the fortunate case that an embedded real-time systems project can choose the hardware to use, and predictable timing and behavior is an issue, the hardware selection should be oriented towards predictability (and thus analyzability).

Our ability to predict the behavior of a real-time system depends on being able to model the behavior of the processors used in sufficient detail, and this appears not to be the case for current high-end processors. Unless a precise simulator can be constructed, we should consider a processor as being unpredictable and unreliable. Thus, selecting a processor which is simple enough to be understood seems prudent for hard real-time systems where predictability is key.

The conventional wisdom is that execution time unpredictability at the hardware level makes superscalar architectures inherently unsuitable for hard real-time tasks where the maximum performance of the processor is required to fulfill requirements [Eyr01], and we agree with this view, especially since performing WCET analysis for such beasts is very difficult.

Another system architecture that is becoming popular but which causes unpredictability in a system is the use of virtual machines like the Java virtual machine (JVM) and binary-to-binary translation schemes like Transmeta's Crusoe [Kla00]. In those systems, the program object code is translated by a just-in-time (JIT) compiler before running on the target hardware [AEGS01]. Such systems make off-line prediction of real-time behavior almost impossible, since both the actual hardware platform used, the final form of the object code, and the overheads of dynamic translation are unknown before the program runs on the target.

Considering the memory system, there are two issues: whether to use a cache, and if a cache is used, how should it should be organized. If variability in execution time has to be minimized, using on-chip fast SRAM under program control instead of a cache is highly appropriate. It should be noted that the programmer-controlled SRAM model has traditionally been employed in DSPs, where being able to predict the performance is paramount [Eyr01]. ARM Ltd. recommends this configuration for timing-critical real-time applications, in the form of the ARM966E-S macrocell [ARM01]. Some research has been performed on how to selectively statically map parts of a program and its data onto on-chip memory [BSL$^+$01, SvP01, ABS01], and such techniques can be used to reduce the manual work required to gain the most from the SRAM.

In many cases, however, there is a strong incentive to use a cache, since this removes the effort to map code and data to different memories. For instruction caches, it is recommended to stick with a direct-mapped cache or a cache with low associativity, since they behave in an analyzable manner. More complex cache organizations should be throroughly checked to ensure that they can be analyzed safely. It is hard to account for the full speed-up achieved by data caches, and thus they might not make the WCET estimate for a program much lower than not having a data cache at all. Unified caches introduce dangerous unpredictability and interference into a system, and are not suitable for real-time systems.

In many cases, choosing a predictable architecture with low inherent variation and no cache is not going to lose much performance compared to a more complex and unpredictable chip, when we have to consider the worst case. As an example, let's investigate the effect of a cache on execution time variation, by a quick back-of-an-envelope calculation: consider a 500 Mhz simple superscalar machine that can execute at most two instructions per cycle, and that has separate data and instruction caches. When every instruction hits the cache, and the code is well-scheduled, we reach a peak cycles-per-instruction value (CPI) of 0.5, giving a performance of 1000 Million instructions per second (MIPS). However, we have to account for the instruction cache behavior. Assume that the cost of a cache miss is twenty cycles (due to the need to refill whole cache lines and the requirement to first detect a cache miss before starting to fetch the missing data to the cache).

The pessimistic scenario that has to be used in the absence of cache analysis is that every instruction fetch and data access misses the cache. If we assume that one in five instructions is a memory access, this brings the CPI down drastically: $CPI = 20 + 20\% * 20 = 25$, corresponding to about 20 MIPS. For a more realistic model, assume that we execute only straight line code, and that we pack four instructions per cache line. In this case, one in four instructions will miss the cache, and we allow two instructions in each cache line to be issued in parallel: $CPI = 25\% * 20 + 25\% * 1 + 50\% * 0.5 + 20\% * 20 = 9.5$, for about 50 MIPS. Thus, we see that the performance of the machine that can be assumed in a safe worst case can be an order of magnitude smaller than the peak performance, effectively wasting the excess processing power.

For comparison, consider a machine that is scalar, runs at 100 Mhz, and has a two-cycle latency local memory. The CPI will be $2 + 20\% * 2 = 2.4$, which translates to about 42 MIPS. This machine will exhibit no variation due to the cache, so this performance can be relied on, reducing the overdimensioning of the system. While the peak speed of the processor is only about 4% of the high-end processor, the worst-case speed (which has to be assumed for a hard real-time system) is almost the same.

Halang and Colnaric [HC97] argue that due to the complexity of modern microprocessors, safety-critical real-time control systems are better implemented using simple and predictable hardware. Several simple processors should be used, with tasks dedicated to each to avoid sources of unpredictability like interrupts disturbing a computational task. This is a reasonable conclusion.

However, many systems are only partially hard real-time, and a fast average-case-optimized processor is used in order to maximize the features available as soft real-time or non-real-time tasks. In such a system, guaranteeing the behavior of the small hard real-time part requires special measures to minimize the variability in execution time (for example, critical code could be locked in the cache). Ideally, the hard real-time work should be off-loaded onto a dedicated processor, but it is not always possible to build ideal systems. More research is needed on how to build partially hard real-time systems using components optimized for average-case performance.

10.4 The Future of WCET Analysis Tools

In the future, we hope to see WCET tools emerge on the mainstream embedded systems development tools market. The most likely scenario is for such tools to be bundled with either compiler suites or support tools for real-time operating systems. The most natural partner is the compiler, since compilers generate a large amount of information that can be very useful for WCET analysis, but which is not part of standard object-code formats and thus cannot easily be communicated to stand-alone tools. Tight integration with a compiler has many practical advantages.

Today, there is no market for static WCET analysis tools, since the only tools available are niche products or research prototypes, and developers are in general unaware of the potential benefits of using static WCET analysis. When a market develops, it is possible that the main use for WCET tool will be for other purposes than scheduling and schedulability analysis, since most real-time systems are not built using operating systems that really support advanced schedulability analysis. Timing analysis of DSP kernels and interrupt latencies can potentially be a much larger market than the analysis of task execution times for the purpose of scheduling.

The development of ever more complex processors poses a long-term challenge for static WCET analysis tools, since even the definition and usefulness of worst-case estimates are problematic given hardware with great enough variability. When the worst case and average case differ by an order of magnitude, it is hard to motivate dimensioning a system for a worst case that will probably "never" occur.

For systems where predictability and guaranteed correctness are important, we hope that system designers realize that predictability has to be designed in from the ground up, starting with the hardware design and choice of processors. The currently available processors for embedded systems are in general analyzable, since processors are kept simple from the dual constraints of power consumption and price. Sufficient performance is enough, and few embedded systems can spare the power and money to use current high-end processors. An encouraging trend is that several simple processors are used in cases where more performance is needed, but power consumption still needs to be kept down [GLN01, Lev01].

It is also a fact that for many real-time applications, the power of a simple 8-bit microprocessor will suffice for the foreseeable future. The number of 32-bit processors sold is increasing very rapidly, but the number of older 8-bit processors remain remarkably steady. Many new applications and products require the use of high-performance processors, but many applications, old and new, use simpler processors since they do not need the performance of high-end processors. All in all, the current embedded systems landscape is rather hopeful for the doability of static WCET analysis.

What is needed today is not so much more research as the practical development of useable tool, and that some development tool vendor dares to take the plunge and test the market for a WCET analysis tool by bringing one to market. On the demand side, students and real-time software developers need to be educated about the benefits of WCET analysis and WCET analysis tools, in order to create a demand for tools.

Bibliography

[ABD$^+$95] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority preemptive scheduling: an historical perspective. *Journal of Real-Time Systems*, 8(2/3):129–154, 1995.

[ABS01] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proc. 4th International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001)*. ACM Press, November 2001.

[AEGS01] Erik R. Altman, Kemal Ebcioglu, Michael Gschwind, and Sumedh Sathaye. Advances and Future Challenges in Dynamic Compilation. *Proceedings of the IEEE*, 89(11), November 2001.

[AKP01] Pavel Atanassov, Raimund Kirner, and Peter Puschner. Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis. In *Proc. IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS 2001*, December 2001.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2), February 2002.

[Alt96] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proc. of the 8th Euromicro Workshop of Real-Time Systems*. IEEE Computer Society Press, 1996.

[AMD00] AMD. *AMD Athlon Processor Architecture*, August 2000.

[AP01] Pavel Atanassov and Peter Puschner. Impact of DRAM Refresh on the Execution Time of Real-Time Tasks. In *Proc. of the Int'l Workshop on Application of Reliable Computing and Communication (WARCC), held along with the IEEE 2001 Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, December 2001.

[ARC] ARC Cores Homepage. `www.arccores.com`.

[ARM95] ARM Ltd. *ARM 7TDMI Data Sheet*, August 1995. Document no. DDI 0029E.

[ARM98a] ARM Ltd. *Application Note 51: ARMulator Cache Models*, January 1998. Document no. DAI 0051A.

[ARM98b] ARM Ltd. *ARM 710T Data Sheet*, July 1998. Document no. DDI 0086B.

[ARM98c] ARM Ltd. *ARM 740T Data Sheet*, February 1998. Document no. DDI 0008E.

[ARM99] ARM Ltd. *ARM 9E-S Technical Reference Manual*, December 1999. Document no. DDI 0165A.

[ARM00a] ARM Ltd. *ARM 720T Data Sheet*, 3rd edition, September 2000. Document no. DDI 0192A.

[ARM00b] ARM Ltd. *ARM 9TDMI Technical Reference Manual*, 3rd edition, March 2000. Document no. DDI 0180A.

[ARM01] ARM Ltd. *ARM9E-S Flyer*, 2001. Document no. D0I 00798A.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. Generally known as the "Dragon Book".

[BA01] M. D. Bennet and N. C. Audsley. Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, June 2001.

[BBMP00] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, pages 39–48. IEEE Computer Society Press, December 2000.

[BC98] Pradip Bose and Thomas M. Conte. Performance Analysis and Its Impact on Design. *IEEE Computer*, 31(5):41–49, May 1998.

[BCF+01] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang, , and Bruce Jacob. The Performance and Energy Consumption of Three Embedded Real-Time Operating Systems. In *Proc. 4th International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001)*. ACM Press, November 2001.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: A Model-Checking Tool for Real-Time Systems. In *Proc. of the 10th International Conference on Computer Aided Verification*, pages 546–550. Springer-Verlag, 1998. LNCS 1427.

[BE00] Alan Burns and Stewart Edgar. Predicting Computation Time for Advanced Processor Architectures. In *Proc. 12th Euromicro Conference of Real-Time Systems, (ECRTS'00)*. IEEE Computer Society Press, June 2000.

[BKY98] Frank Burns, Albert Koelmans, and Alexandre Yakovlev. Analysing superscalar processor architectures with coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:182–191, 1998.

[Bör95] Hans Börjesson. Incorporating worst-case execution time in a commercial c-compiler. Master's thesis, Department of Computer Systems, Uppsala University, December 1995. DoCS MSc Thesis 95/69.

[BPPS00] Valerie Bertin, Michel Poize, Jacques Pulou, and Joseph Sifakis. Towards validated real-time software. In *Proc. 12th Euromicro Conference of Real-Time Systems, (ECRTS'00)*. IEEE Computer Society Press, June 2000.

[BS98] Bryan Black and John Paul Shen. Calibration of Microprocessor Performance Models. *IEEE Computer*, 31(5):59–65, May 1998.

[BSL+01] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Comparison of Cache- and Scratch-Pad based Memory Systems with respect to Performance, Area, and Energy Consumption. Technical Report #762, Universität Dortmund, Lehrstuhl Informatik XII, August 2001.

[Car02] Martin Carlsson. WCET Analysis of Interrupt-Disable Regions in Enea OSE. Master's thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, January 2002. To appear.

[CBW94] R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94)*, 1994.

[Ced02] Gittan Cedervall. PowerPC växer på fallande marknad. *Elektroniktidningen*, February 15, 2002.

[Cha95] R. Chapman. *Static Timing Analysis and Program Proof.* PhD thesis, Department of Computer Science, University of York, England, 1995.

[CHK$^+$96] Kenneth K. Chan, Cyrus C. Hay, John R. Keller, Gordon P. Kurpanek, Francis X. Schumacher, and Jason Zheng. Design of the HP-PA 7200 CPU. *Hewlett-Packard Journal*, February 1996.

[Col00] Chris Collins. An Evaluation of Embedded System Behavior Using Full-System Software Emulation. Master's thesis, University of Maryland, College Park, 2000.

[Com99] Compaq. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999. Document Number EC-RJRZA-TE.

[CP00] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, 18(2/3):249–274, May 2000.

[CP01a] A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01).* IEEE Computer Society Press, June 2001.

[CP01b] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01).* IEEE Computer Society Press, June 2001.

[CP01c] Antoine Colin and Isabelle Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. Technical Report Publication Interne No 1386, IRISA, March 2001.

[CRTM98] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – a revolution in on-board communications. *Volvo Technology Report*, 1:9–19, 1998.

[Dav71] E. S. Davidson. The design and control of pipelined function generators. In *Proc. 1971 International IEEE Conference on Systems, Networks, and Computer (SNC)*, pages 19–21, January 1971.

[DBK01] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *Proc. of the 28th International Symposium on Computer Architecture (ISCA 2001).* ACM Press, July 2001.

[DEC98] Digital Equipment Corporation. *Alpha 21164 Microprocessor Data Sheet*, August 1998. Document Number EC-QP98C-TE.

[DS99] A. Dean and J. P. Shen. System-Level Issues for Software Thread Integration: Guest Triggering and Host Selection. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99)*, 1999.

[EB01] Stewart Edgar and Alan Burns. Statistical Analysis of WCET for Scheduling. In *Proc. 22st IEEE Real-Time Systems Symposium (RTSS'01)*, December 2001.

[EE99] Jakob Engblom and Andreas Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99).* IEEE Computer Society Press, December 1999.

[EE00] Jakob Engblom and Andreas Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[EES$^+$99] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Towards industry-strength worst case execution time analysis. Technical Report ASTEC 99/02, Advanced Software Technology Center (ASTEC), April 1999.

[EES00] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. Comparing Different Worst-Case Execution Time Analysis Methods. In *Proc. of the Work-in-Progress Session at the 21st Real-Time Systems Symposium (RTSS/WIP 2000)*, December 2000.

[EES01a] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems. In *Proceedings of the Workshop on Real-Time Tools (RT-TOOLS 2001), held in conjunction with CONCUR 2001*, August 2001.

[EES01b] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. Validating a Worst-Case Execution Time Analysis Method for an Embedded Processor. Technical Report 2001-030, Dept. of Information Technology, Uppsala University, 2001. `www.it.uu.se/research/reports/2001-030/`.

[EG97a] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer-Verlag, August 1997.

[EG97b] Andreas Ermedahl and Jan Gustafsson. Realtidsindustrins syn på verktyg för exekveringstidsanalys. Technical Report ASTEC Technical Report 97/06, ASTEC Competence Center, Uppsala University, July 1997. `www.astec.uu.se/Reports/Reports/9706.ps.gz`.

[Eng97] Jakob Engblom. Worst-Case Execution Time Analysis for Optimized Code. Master's thesis, Department of Computer Systems, Uppsala University, September 1997. DoCS MSc Thesis 97/94.

[Eng98] Jakob Engblom. Static Properties of Commercial Real-Time and Embedded Systems. Technical Report DoCS 98/102, Dept. of Computer Systems, Uppsala University, November 1998. `www.docs.uu.se/docs/research/reports/`.

[Eng99a] Jakob Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.

[Eng99b] Jakob Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

[ESE00] Jakob Engblom, Friedhelm Stappert, and Andreas Ermedahl. Structured Testing of Worst-Case Execution Time Analysis Tools. In *Proc. of the Work-in-Progress Session at the 21st Real-Time Systems Symposium (RTSS/WIP 2000)*, December 2000.

[ESS99] E. Erpenbach, F. Stappert, and J. Stroop. Compilation and Timing Analysis of Statecharts Models for Embedded Systems. In *Proc. 2nd International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*. ACM Press, October 1999.

[EY97] R. Ernst and W. Ye. Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification. In *International Conference on Computer-Aided Design (ICCAD '97)*, 1997.

[Eyr01] Jennifer Eyre. The Digital Signal Processor Derby. *IEEE Spectrum*, 38(6), June 2001.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *Proc. 5th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS92)*, 1992.

[FHL$^+$01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proc. First International Workshop on Embedded Software (EMSOFT 2001), LNCS 2211*. Springer-Verlag, October 2001.

[FKL$^+$00] Christian Ferdinand, Daniel Kästner, Marc Langenbach, Florian Martin, Michael Schmidt, Jörn Schneider, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Run-Time Guarantees for Real-Time Systems– The USES Approach. In *Proc. of the "Deutschsprachige WCET-Tagung" held at C-Lab in Paderborn in conjunction with DIPES 2000 (International IFIP Workshop on Distributed and Parallel Embedded Systems)*, October 2000. `http://www.c-lab.de/wcet2000/`.

[FLBC01] Scott Friedman, Nicholas Leidenfrost, Benjamin C. Brodie, and Ron K. Cytron. Hashtables for Embedded and Real-Time Systems. In *Proc. IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS 2001*, December 2001.

[FMW97] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[Gan01] Jack Ganssle. Really Real-Time Systems. In *Proc. Embedded Systems Conference San Fransisco (ESC SF) 2001*, April 2001.

[Gan02] Jack Ganssle. Ada's slide into oblivion. "Embedded Pulse" Column on `www.embedded.com`, January 2002.

[GKO$^+$00] Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy, and Mark Heinrich. FLASH vs. (Simulated) FLASH: Closing the Simulation Loop. In *Proc. 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS00)*, November 2000.

[GLN01] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip. In *Proc. 22st IEEE Real-Time Systems Symposium (RTSS'01)*, December 2001.

[Gus00] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000. DoCS Report 00/115, `www.docs.uu.se/docs/research/reports/`.

[Had02] Patric Hadenius. En dator i varje pryl. *Forskning och Framsteg*, (1/02), January-February 2002.

[Hal99] Tom R. Halfhill. StarCore Reveals Its First DSP. *Microprocessor Report*, May 10, 1999.

[Hal00a] Tom R. Halfhill. EEMBC Releases First Benchmarks. *Microprocessor Report*, May 1, 2000.

[Hal00b] Tom R. Halfhill. Embedded Market Breaks New Ground. *Microprocessor Report*, January 17, 2000.

[HAM$^+$99] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[Han98] Jim Handy. *The Cache Memory Book.* Academic Press, 2nd edition, 1998.

[HC97] Wolfgang A. Halang and Matjaz Colnaric. On Safety-Critical Computer Control Systems. In *Proc. Tenth IEEE Symposium on Computer-Based Medical Systems*, June 1997.

[Hei94] J. Heinrich. *MIPS R4000 Microprocessor User's Manual.* MIPS Technologies Inc., 2nd edition, 1994.

[Her98] Stephen A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior.* PhD thesis, Stanford University, February 1998.

[HHWT97] T. A. Henzinger, P-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. In *Proc. of the 9th International Conference on Computer Aided Verification*, pages 460–463. Springer-Verlag, 1997. LNCS 1254.

[Hit95] Hitachi Europe Ltd. *SH7700 Series Programming Manual*, September 1995.

[HKA+01] Christopher J. Hughes, Praful Kaul, Sarita V. Adve, Rohit Jain, Chanik Park, , and Jayanth Srinivasan. Variability in the Execution of Multimedia Applications and Implications for Architecture. In *Proc. of the 28th International Symposium on Computer Architecture (ISCA 2001)*. ACM Press, July 2001.

[HLS00a] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Using a Worst-Case Execution-Time Tool for Real-Time Verification of the DEBIE software. In *Proceedings of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, September 2000.

[HLS00b] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, September 2000.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2nd edition, 1996. ISBN 1-55860-329-8.

[HPRA02] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, 35(2), February 2002.

[HSR+00] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting Timing Analysis by Automatic Bounding of Loop Iterations. *Journal of Real-Time Systems*, 18(2/3):129–156, May 2000.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[HW99] C. Healy and D. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, June 1999.

[IAR99] IAR Systems. *V850 C/EC++ Compiler Programming Guide*, 1st edition, January 1999.

[Inf01] Infineon. *Instruction Set Manual for the C166 Family*, 2nd edition, March 2001.

[Int00a] Intel. *Intel XScale Core Developer's Manual*, December 2000.

[Int00b] Intel. *Itanium Processor Microarchitecture Reference for Software Optimization*, August 2000. Document Number: 245473-002.

[Int01a] Intel. *Intel Pentium 4 and Intel Xeon Processor Optimization*, 2001. Document Number: 248966-04.

[Int01b] Intel. *Intel Pentium 4 Processor Specification Update*, April 2001.

[Ive98] Anders Ive. Runtime Performance Evaluation of Embedded Software. In *Presented at the Eighth Nordic Workshop on Programming Environment Research*, August 1998. `http://www.ifi.uib.no/konf/nwper98/proceedings.html`.

[KHM99] Sung-Kwan Kim, Rhan Ha, and Sang Lyul Min. Analysis of the Impacts of Overestimation Sources on the Accuracy of Worst Case Timing Analysis. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99)*, December 1999.

[Kir02] Raimund Kirner. The Programming Language WcetC. Technical Report 2/2002, Technische Universität Wien, Institut für Technische Informatik, January 2002.

[Kla00] Alexander Klaiber. *The Technology Behind Crusoe Processors (Whitepaper)*. Transmeta Corporation, January 2000.

[KLP01] Raimund Kirner, Roland Lang, and Peter Puschner. WCET Analysis for Systems Modeled in Matlab/Simulink. In *Proc. 22st IEEE Real-Time Systems Symposium (RTSS'01), Work in Progress Session*, December 2001.

[KMH96] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 230–240, 1996.

[Kog81] Peter M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill Book Company, New York, N. Y., 1981.

[KP01] Raimund Kirner and Peter Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, June 2001.

[Kum97] A. Kumar. The HP PA-8000 RISC CPU. *IEEE Micro*, 17(2), March 1997.

[LBJ$^+$95] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[Lev01] Markus Levy. ManArray Devours DSP Code. *Microprocessor Report*, October 8, 2001.

[LG98] Y. A. Liu and G. Gomez. Automatic Accurate Time-Bound Analysis for High-Level Languages. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, pages 31–40, June 1998.

[LH02] Gabriel Leen and Donal Hefferman. Expanding automotive electronic systems. *IEEE Computer*, 35(1), January 2002.

[LHKM98] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[LKM98] S-S. Lim, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Optimized Programs. In *Proc. 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, pages 151–157. IEEE Computer Society Press, Oct 1998.

[LL73] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LM95] Y-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd ACM IEEE Design Automation Conference (DAC'95)*, pages 456–461, 1995.

[LMW96] Y-T. S. Li, S. Malik, and A. Wolfe. Cache modelling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263. IEEE Computer Society Press, December 1996.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, December 1997.

[LS98] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, June 1998.

[LS99a] Thomas Lundqvist and Per Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Journal of Real-Time Systems*, 17(2/3):183–207, November 1999.

[LS99b] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99)*, December 1999.

[Mär01] Andreas Märcz. Scheduling periodic media converters. Presented at the Workshop on the Future of WCET Analysis held in conjunction with the Euromicro Real-Time Systems Conference 2001, June 2001.

[Mon00] Sven Montán. Validation of Cycle-Accurate CPU Simulator against Actual Hardware. Master's thesis, Dept. of Information Technology, Uppsala University, 2000. Technical Report 2001-007, `http://www.it.uu.se/research/reports/2001-007/`.

[Mot90] Motorola Inc. *MC88200 Cache/Memory Management Unit User's Manual*, 1^{st} edition, 1990.

[Mot97] Motorola Inc. *MPC750 RISC Microprocessor User's Manual*, August 1997.

[Mot01] Motorola Inc. *MPC850 Family Device Errata Reference*, February 2001.

[MR01] Tulika Mitra and Abhik Roychoudhury. Effects of Branch Prediction on Worst Case Execution Time of Programs. Technical Report 11-01, National University of Singapore (NUS), November 2001.

[Muc97] S. S. Muchnick. *Advanced Compiler Design.* Morgan Kaufmann Publishers, 1997.

[Mue97] Frank Mueller. Timing predictions for multi-level caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, Jun 1997.

[Mue00] Frank Mueller. Timing Analysis for Instruction Caches. *Journal of Real-Time Systems*, 18(2/3):209–239, May 2000.

[NEC95] NEC Corporation. *V850 Family 32/16-bit Single Chip Microcontroller User's Manual: Architecture*, 4^{th} edition, 1995. Document no. U10243EJ4V0UM00.

[NEC99] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3^{rd} edition, January 1999. Document no. U12197EJ3V0UM00.

[Nil01] Magnus Nilsson. Time Accurate Simulation. Master's thesis, Dept. of Information Technology, Uppsala University, September 2001. Thesis number: UPTEC F 01 074, `www.astec.uu.se/Reports/Reports/nilsson-01-tas.pdf`.

[OS97] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Journal of Real-Time Systems*, 5(1):31–62, March 1993.

[Pat95] Jason R. C. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, June 1995.

[Pat01] Yale Patt. Requirements, bottlenecks, and good fortune: Agents for microprocessor evolution. *Proceedings of the IEEE*, 89(11), November 2001.

[PB01] Peter Puschner and Guillame Bernat. WCET Analysis of Reusable Portable Code. In *Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01).* IEEE Computer Society Press, June 2001.

[Pet00] Stefan Petters. Bounding the Execution Time of Real-Time Tasks on Modern Processors. In *Proc. 7^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'00).* IEEE Computer Society Press, December 2000.

[PF99] S. Petters and G. Färber. Making Worst-Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99).* IEEE Computer Society Press, December 1999.

[PK89] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(1):159–176, 1989.

[PPVZ92] Gustav Pospischil, Peter Puschner, Alexander Vrchoticky, and Ralph Zainlinger. Developing Real-Time Tasks With Predictable Timing. *IEEE Software*, 9(5), September 1992.

[Pri95] Dick Price. Pentium FDIV flaw–lessons learned. *IEEE Micro*, April 1995.

[PS90] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. In *Proc. 11th IEEE Real-Time Systems Symposium (RTSS'90)*, pages 72–81, December 1990.

[PS95] Peter Puschner and Anton Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[Ram00] G. Ramalingam. On Loops, Dominators, and Dominance Frontier. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2000)*, June 2000.

[Ray00] E. Raymond. The jargon file, version 4.2.0. `http://www.tuxedo.org/~esr/jargon/html/index.html`, February 2000.

[SA00] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[SEE01a] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. Technical Report 2001-012, Dept. of Information Technology, Uppsala University, 2001. `www.it.uu.se/research/reports/2001-012/`.

[SEE01b] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4th International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001)*. ACM Press, November 2001.

[SF99] Jörn Schneider and Christian Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

[SFK97] Dezsö Sima, Terence Fountain, and Péter Kacsuk. *Advanced Computer Architectures: A Design Space Approach*. Addison-Wesley, 1997.

[SGL96] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.

[SKO$^+$96] V. Seppänen, A-M Kähkönen, M. Oivo, H. Perunka, P. Isomursu, and P. Pulli. Strategic needs and future trends of embedded software. Technical Report Technology Review 48/96, TEKES Technology Development Center, Oulu, Finland, October 1996.

[Sny01] Cary D. Snyder. Synhesizable Core Makeover–Is Lexra's Seven-Stage Pipelined Core the Speed King? *Microprocessor Report*, February 7, 2001.

[Son97] Peter Song. UltraSparc-3 Aims at MP Servers. *Microprocessor Report*, October 27, 1997.

[Sta97] Friedhelm Stappert. Predicting pipelining and caching behaviour of hard real-time programs. In *Proc. of the 9th Euromicro Workshop of Real-Time Systems*. IEEE Computer Society Press, June 1997.

[Ste01] David B. Stewart. Twenty-Five Most Common Mistakes with Real-Time Software Development. In *Proc. Embedded Systems Conference San Fransisco (ESC SF) 2001*, April 2001.

[SvP01] Jan Sjödin and Carl von Platen. Storage Allocation for Embedded Processors. In *Proc. 4th International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001).* ACM Press, November 2001.

[Ten99] David Tennenhouse (Intel Director of Research). Keynote Speech at the 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona, December 1999.

[TF98] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[TI00] Texas Instruments Inc. *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000. Literature Number: SPRU189F.

[Tur97] Jim Turley. ARM9 Doubles ARM Performance in '98. *Microprocessor Report*, December 8, 1997.

[VHMW01] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric Timing Analysis. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'2001)*, June 2001.

[WB98] Ole Wolf and Jeff Bier. TigerSharc Sinks Teeth Into VLIW. *Microprocessor Report*, December 7, 1998.

[WE01] F. Wolf and R. Ernst. 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*, 47(3-4):339–356, April 2001.

[WMH$^+$97] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, June 1997.

[ZWR$^+$01] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst. Interval-Based Analysis of Software Processes. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'2001)*, June 2001.