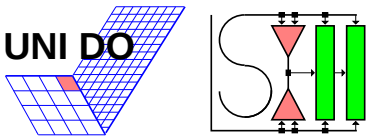
Master's Thesis

Design and Realization of Concepts for WCET Compiler Optimization

Paul Lokuciejewski



University of Dortmund
Department of Computer Science 12

December 22, 2005

Advisors:

Dr. Heiko Falk
Prof. Dr. Peter Marwedel

Contents

List of Figures

List of Tables

Preface

I would like to thank my advisor Dr. Heiko Falk for his enthusiasm and excellent support during this thesis. He provided a lot of valuable input and comments on this research. Further, I want to direct many thanks to Robert Pyka for our interesting discussions.

The employees at AbsInt Angewandte Informatik GmbH also deserve my gratitude. I especially want to thank Dr. Henrik Theiling for answering all my questions about the aiT WCET analyzer. His enthusiasm has been very supportive. I would also like to thank Michael Schmidt having answered my questions about aiT.

I want to direct many thanks to my friends Kirill Perfiliev, Llyod Winn and Christoph Heckmann. Thank you for taking the time to read and comment this thesis.

Finally, I want to express my gratitude to my parents Marian and Elisabeth, and my sister Anna, for your support and interest in my work. Thank you.

Dortmund, December 22, 2005

Chapter 1

Introduction

1.1 Why Compilers for Embedded Systems?

With each day, our lives become more and more dependant on digital systems. A landslide majority of today's private households possess a computer which is used for a vast variety of tasks. However, these gadgets represent the modest stake in this domain. *Embedded Systems* are the information processing part of a larger product that are found more frequently. They are digital systems which carry out their services "invisibly". Studies expose that we are in permanent interactions with digital systems. Andy Grove, an Intel co-founder, recently speculated that an average American comes into contact with 72 microprocessors before lunch. Mobile phones or automobiles like the Mercedes S-class with over 100 microprocessors are just some prominent examples. We are living in the third era of computing. After mainframes and personal computers in the first and second era, respectively, nowadays the digital technology recedes in the background. Thus, it is justified to call this period the period of *Ubiquitous computing* [Mar06].

To sum it up, digital systems are divided into two sectors, namely the *general-purpose systems* and the *special-purpose systems (embedded systems)*. The former are traditional computers covering ordinary PCs, servers and super-computers. Their common property is that they can be programmed by a user and can be deployed for a large number of different applications. But nowadays, new applications come into being which demand characteristics that can not be provided by general-purpose systems for technical reasons.

To meet the requisites, special-purpose systems were developed. They are programmed once and provide service for a specific task without any further interaction with the user. Due to the demand of new and more efficient applications, embedded systems gain in importance. In 2001, 4bn US $ were spent on embedded processors while in 2005, the volume doubled occupying 95% of the entire processor market [Pet05].

Besides the already mentioned fields of application, special-purpose systems are strongly used in the following areas:

- Multimedia: DVD/MP3/CD players, video games, digital cameras, digital television
- Telecommunication: Internet routers, UMTS, speech codes, wireless protocols
- Automotive engineering: Engine control, automatic transmission, ABS, GPS, ESP
- Control applications: industrial process controllers
- Medical systems: heart pacemakers

In contrast to general-purpose systems, embedded systems usually have to meet some real-time constraints making them real-time systems. The correctness of a real-time system is by definition given as follows [BW01]:

> The correctness of a real-time system depends not only on the logical result of the computation but also on the time at which the results are produced.

In addition to the need of the obvious constraint of logical correctness, these time-critical systems depend upon the point of time a result is generated. Especially in control applications where these computer systems are deployed to control and steer some essential processes, a failure in providing the mandatory results on time can entail physical damage, injury or in worst case loss of human life.

The importance of the time factor allows a distinction between those systems which will suffer a failure if timing constraints are violated (hard real-time systems) and those which will not (soft real-time systems). *Hard real-time systems* strictly rely on the amount of time a program requires to finish. Missed deadlines make the delayed results useless and eventually prevent from continuing the program in a correct fashion. As an example, a car engine control system is a hard real-time system due to the fact that a delayed signal may wreck the engine. On the other hand, *soft real-time systems* are used in the area where concurrent access is an issue, but a missed deadline does not prevent the system from proceeding. For instance, a delayed signal in a live audio system results in a degraded quality but does not cause any damage.

Furthermore, the market demands high performance, energy efficient and low cost products making the time factor an important issue. The knowledge of the worst-case execution time (WCET), which will be explained in detail in section 1.3, gives the real-time system engineer the opportunity to use or design a hardware platform which is tailored towards the software resource requirements like memory or clock rate. Thus, the production costs can be slashed. Further advantages emerging from having the WCET available can be found in the area of simulation and verification of time critical systems [EEN+99].

To meet the aforementioned efficiency-criteria, programmers used to write their code in the assembly language. However, rising complexity of applications and the desire for reusability of programs and program libraries shifted the focus away from assembly programming towards high-level language programming, mainly C [ISO99] and C++ [ISO98]. Driven by this circumstance, the task to generate optimized and efficient machine

code was moved from the programmer to a compiler creating new challenges for compiler constructors.

1.2 Compiler Optimization

A compiler is a computer program that translates a program written in one computer language (source language) into a semantically equal program written in another language (target language). In the domain of embedded systems, C and C++ are predominantly used as source language. Due to the need for highly efficient machine code, compilers become the key component in an integrated development environment.

Compiler phases
A state of the art compiler shares a two-stage design consisting of a frontend and a backend. Within the initial phase, the *frontend*, the source code provided as input is analyzed and translated into a medium-level intermediate representation. The second phase is the *backend* which works with the low-level intermediate representation of the code to produce output in the target language, mainly the optimized machine code. The frontend consists of multiple phases:

- **Lexical Analyzer**
  A scanner reads an input string of characters (source code) and generates a sequence of symbols (so called lexical tokens).
- **Parser**
  The syntactical structure of the source code is identified by a parser and transformed into a hierarchical structure, the parse tree.
- **Semantic Analyzer**
  This phase verifies the meaning of the input and adds semantical information to the parse tree. Based on this data, various checks are performed and compiler errors issued.
- **IR-Code Generator**
  Finally, the parse tree is read and translated into a medium-level representation.

In the next phase, the compiler passes the intermediate representation generated by the frontend to the backend, which uses the code generator to translate the low-level intermediate representation into the output language. Figure 1.1 gives a pictorial view of the phases ran through by an optimizing compiler.

Optimizations
Besides the mandatory steps, a compiler performs multiple optimizations both processor-independent and processor-dependent. Examples for the former which involve the medium-level intermediate representation are:

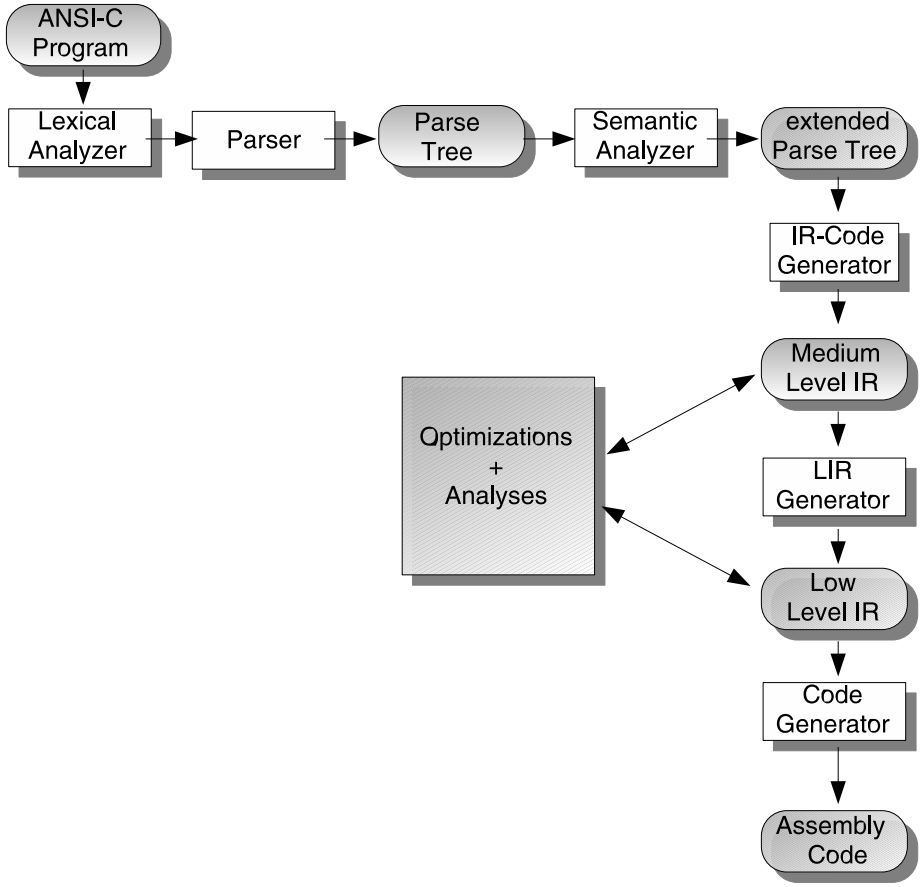


Figure 1.1: Structure of an optimizing compiler

- **Inline expansion**
  For short and frequently used functions, it is beneficial to substitute the function call directly by the function body. Thus, the overhead associated with the function call is reduced.

- **Dead code elimination**
  Any portion of program code which produces values that are never used at any point of the program, can be removed in order to reduce code size.

```
int foo() {
  int a = 1;
  int b;
  return a;
  b = 2;
}
```

  The assignment of the value `2` to the variable `b` can be removed since the code is left after the return statement.

- **Constant folding**
  Constant expressions can be simplified by shifting their calculation from runtime to compile time resulting in a reduced number of used instructions.

  ```
  int a = 640 * 480 * 2;
  ```

  A modern compiler would reduce the statement to `int a = 614400` instead of using two multiply instructions.

- **Loop-invariant code motion**
  Computations which can be moved before or after a loop without affecting the semantics of the program are called loop-invariant. This optimization results in a smaller code providing a speed-up.

  ```
  while ( i <= a + 5 ) {
    // a is not modified here
    ...
  }
  ```

  Since the variable `a` is not changed in this expression, it can be placed before the loop:

  ```
  t = a + 5;
  while ( i <= t )
  ```

In the next step, the backend translates the optimized code to a low-level intermediate representation which is used by the code generator to produce code in the output language. This is done in multiple steps comprising the performance of processor-dependent optimization techniques. Popular optimizations are:

- **Code selection**
  The code selection phase replaces operations from the low-level intermediate representation by suitable assembly instructions. While selecting instructions, a cost function is considered that is meant to be minimized.

- **Register allocation**
  Due to the fact that the number of physical registers is limited to a small fixed number, this phase is mandatory. The register allocator decides whether operations are to be performed on registers or in memory with the aim to make the execution of a program as fast as possible. Usually, this is done by minimizing the number of load and store instructions between the memory and registers. The most popular approaches for this NP-complete problem used in modern compilers are the graph coloring algorithm [Cha82] and Integer Programming algorithms [GW96].

- **Instruction scheduling**
  To minimize the execution time of a program, instruction-level parallelism is exploited in the instruction scheduling phase. Taking instruction pipelines into ac-

count, the order of instructions is rearranged so that parallel-working units are utilized without violating dependencies.

The optimization techniques mentioned above are just a small example of what modern compilers are dealing with. For instance, a state of the art compiler, like the GNU C Compiler, comprises more than forty optimizations which run either automatically or after a user interaction [GCC05].

Typical concepts that present-day compilers focus on are the average time (also known as average-case execution time, ACET) or the energy awareness. The latter was successfully realized in the *encc* compiler which was developed at the Embedded Systems Groups of the Computer Science Department at Dortmund University [SW05]. Despite the importance of the worst-case execution time, there have not been any successful integrations of the WCET within a compiler yet.

1.3 Compiler-Integration of Worst-Case Execution Time (WCET)

By definition, the worst-case execution time is the longest execution time of a program run on a specific hardware [Erm03]. This definition assumes that the program is executed on an isolated system which is not disturbed by any external interactions like operation system activities or user input. A related problem is the best-case execution time (BCET) which is the shortest execution time of a program ever observed on a given hardware. Finally, the average-case execution time (ACET) is an execution time lying somewhere between the WCET and the BCET. The latter is typically the issue a home desktop-PC is optimized for. Due to the fact that home computers do not rely on a guarantee to generate a signal in a specific amount of time, they do not belong to the domain of real-time systems. On a typical multimedia system running a number of applications in parallel, the limited resources are not able to permit all processes to be executed in the optimal amount of time. Thus, the most convenient way for a user to optimize all running applications in terms of their average execution time is to reduce latency.

Timing analyses can not guarantee to evaluate the exact program execution time. Rather, they aim to produce an estimation of the actual execution time in a reasonable amount of time while consuming a reasonable amount of resources. There are two constraints to be met in order to receive a useful time estimation:

- **Safeness**
  A timing estimate is meant to be safe when it is not underestimated in terms of the the actual execution time. It states the reliability of the the result and is an essential condition.
- **Tightness**

The tighter the estimate, with respect to the actual execution time, the more realistic the results are. Tightness describes the overestimation resulting from the deviation between the real and the estimated execution time.

Applying these conditions to the WCET and BCET obviously results in an oppositional situation. In terms of the worst-case execution time, an estimate is safe when it is larger than the actual time and it is tight when the difference between the estimate and the actual time is minimal. On the other hand, the BCET estimate is safe when it does not exceed the actual time and it is tighter the smaller the difference between the actual time and the estimate is. Figure 1.2 illustrates the different conditions.

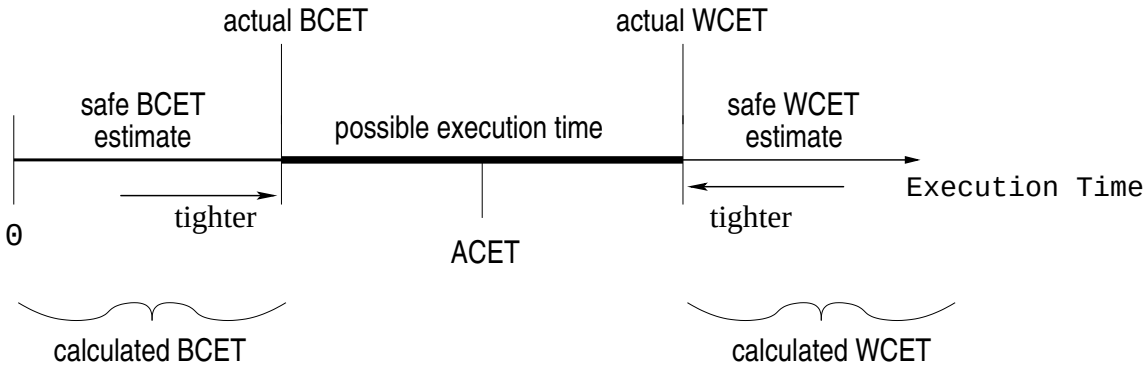


Figure 1.2: Estimations and conditions for the execution time

In both cases, the violation of the safeness makes the estimate useless for a real-time system since it can not be guaranteed to produce results in a given amount of time. The consequences incurred can be of different extent, ranging from a neglecting failure in a module up to a catastrophe that entails physical damage or loss of life. The research activities in the domain of scheduling analyses involving the ACET are nowadays highly sophisticated since they were the subject of study by innumerable scientists in the past going back to the 1970's. In contrast, the problem of estimating the worst-case execution time remained unsolved for a long time. In recent years, the WCET analysis has gained in importance, but is often strongly focused on some specific sets of problems, and few versatile approaches are known yet.

The approaches to generate a WCET estimate are divided into two categories: dynamic analysis and static analysis. The traditional state of the practice is the *dynamic analysis* (also known as measuring) which is running the program to gain the timing data. Within this empirical method, the first task is to find a subset of typical inputs which are supposed to represent the worst case and therefore yield the longest execution time of the program. The measurements are performed by using oscilloscopes or logic analyzers. The advantages are that no additional applications have to be integrated in the design progress, and the system does not need to be thoroughly understood but can be considered as a black box. For this reason, the overhead for new hardware to be analyzed is reduced to the proper integration of the measuring instruments and does not entail the development of new ap-

plications. Nonetheless, the drawbacks overbalance the relatively simple integration of the dynamic analysis. On the one hand, there is no guarantee that the used input data really represents the worst input since, for technical reasons, not all possible combinations of input data can be taken into account. Thus, the security risk remains that the real-time system fails while operating in an authentic environment. On the other hand, data measured by physical instruments is always afflicted with a measurement error leading to a deviation from the actual values. Last but not least, the dynamic analysis requires the target machine. Thus, the performance of the analysis is restricted to the locations the hardware is available at. As a conclusion, it can be said that this approach is sufficient for soft real-time systems that are not designed to operate in environments where a failure or violation of conditions endangers human lives. This approach can also be used for hard real-time systems by multiplying the estimate by some safety factor to guarantee an adequate tolerance in order to always meet the real-time conditions. It does not need to be mentioned that this increase of the WCET is at the expense of hardware resources and therefore rises production costs.

The second class used to obtain a WCET estimate is the *static analysis*. The program is not run, but its structure is analyzed formally using the flow information with the the objective of obtaining the maximum execution path. In contrast to the dynamic analysis, the estimate obtained by this approach ensures to be valid for all possible inputs. Further extensive measurements are redundant and therefore avoid falsification of the results. Consequently, the calculated WCET is guaranteed to be safe and tight reflecting the authentic behavior of the program.

The worst-case execution time of a program depends upon both the hardware used as a platform and the structure of the program. The former involves aspects such as the processor speed, the amount and access time of memory, the bus system etc. In earlier days, the analysis problem was often deemed a designer task, and oversized hardware was chosen to fulfill the needs of the software. This led to a waste of resources but was not an issue as the demands for minimal hardware were not given. With the advent of embedded systems that require a small size and a low power consumption, the focus was shifted away from the hardware choice to the optimization of software resulting in a more efficient code. This enabled the execution of the software on downgraded hardware.

The structure of a program is represented by its *program flow* that describes in which fashion the program code is executed. This involves loop iterations, function calls (and eventually their recursion depth) and code execution concerning `if`-statements, etc. Besides this data, a further phase is mandatory, namely the *low-level analysis*. The goal of this phase is to determine the timing information of each atomic unit that is either executed completely or not at all within an execution path of the program flow. Invariably, atomic units on a low level are represented by so called basic blocks which are comprised of instructions.

Another important aspect, which has to be considered to obtain a precise estimation, is the underlying hardware architecture. A great number of research results within the

field of WCET estimation assume a simplified situation by neglecting hardware features like branch prediction, caches and pipelines. This leads to substantial overestimates of the WCET and thus a waste of resources. Since the research field of cache behavior is an extensive issue, there is often a lack of this information forcing the researchers to elude this hardware information. Older microcontrollers like the popular Intel 8051 guaranteed a fixed execution time for each instruction. To obtain the execution time of a basic block, a simple addition had to be performed. However, modern processors come usually with the aforementioned features enabling the acceleration of the execution time of a program. But on the other hand, they complicate the timing analysis since execution times for instructions are variable. Pipelines of a processor accelerate the throughput of a program by splitting larger instructions into smaller ones and so permit an independent execution. But this also leads to a variable execution time of an instruction which depends on the current situation concerning other instructions processed in parallel. Similarly, a processor unit using cache memory yields variable instruction execution times since the memory access time strongly depends upon the current situation, i.e. whether the data required by the instruction is already stored in the cache memory (known as cache hit) or not and thus has to be fetched from main memory (called cache miss). World-wide, there are just a few vendors of analysis tools which incorporate the consideration of the above mentioned processor features into their tools. One of them is the company AbsInt Angewandte Informatik GmbH [Abs05] located in Saarbrücken, Germany. Their sophisticated tool, the *aiT WCET Analyzer*, is used in this master's thesis as part of the framework. A detailed description will be given in chapter 2.

The last step to calculate the estimated WCET is to combine the information gained from the program flow and the low-level analysis. The flow information indicates parts of the code which are never traversed (infeasible paths). Moreover, it provides information on all potential paths depending on the state of the program which the machine can pass through when the program is executed. Adding the timing information from the low-level analysis to each basic block allows one to find the path which takes the longest time to execute.

Three different calculation methods are widely spread, namely the Implicit Path Enumeration Technique (IPET), the Path-based and the Tree-based calculation method. The *Implicit Path Enumeration Technique (IPET)* calculates the WCET estimate by maximizing an objective function while satisfying the given constraints. Each edge between two basic blocks is assigned a WCET bound t_{edge} giving the contribution of that part of the code to the total execution time. Furthermore, c_{edge} defines the *execution count* representing the number of control passes along *edge* between two basic blocks. The constraints are given by the control structure of the program, e.g. concerning each node, the sum of the incoming flows has to be equal to the sum of outgoing flows. Also, information about infeasible paths imposes additional constraints. Obtaining the global WCET aims at finding the maximum sum of the products of the execution time and the execution counts:

$$global\ WCET = \max(\sum_{\forall edge} t_{edge} \cdot c_{edge})$$

The *Path-based calculation* generates an estimate for the WCET by calculating upper bounds for possible paths in the control flow and finding the one with the longest execution time. To do so, standard graph algorithms like Dijkstra's algorithms are applied [EES00]. Finally, the *Tree-based calculation* employs a syntax tree representing the program whose nodes characterize the structure of the program like loops or conditions and whose leaves describe basic blocks. To generate the final WCET, the syntax tree is traversed bottom-up and each node is transformed into an equation expressing its timing. Each calculation approach has its assets and drawbacks and should be evaluated in order to determine the most efficient one for a particular purpose.

As already mentioned, the increasing number of real-time systems demands information on the worst-case execution time to minimize the risk of failure and shorten the needs of the underlying hardware. It is desirable to integrate a timing analyzer into a compiler, making it an ordinary part of the compilation toolchain akin to modules performing the well known optimization techniques mentioned in section 1.2. Ideally, it should be up to the programmer to steer the compilation progress by easily choosing the WCET optimizations he wants to enable. Possible issues could be:

- **Resource needs**
  Optimizing the code in terms of the worst-case execution time decreases the requirements of the hardware the program is to be run on. Thus, cheaper hardware components may be used and thus cut the production costs.
- **Timing factor**
  The worst-case execution time can be reduced at the expense of other resources. For instance, a dual instruction set processor might be used for a tradeoff between the WCET and the code size. This approach will be explained in more detail in section 1.4.
- **Power Consumption**
  Knowing the WCET estimates, power consumption can be reduced by lowering the clock rate of a processor to such an extent that the real-time constraints still are guaranteed to be met [ZKW+04].

1.4 Related Work

Unlike the ACET, the number of WCET research and development groups is still narrow due to the fact that this research field remained untackled for a long time. However, activities in the last years have shown that the subject of WCET gained in importance. Evidence of this is a wish list [BH03] which is addressed to compiler vendors with the objective of redesigning their software to provide data useful for timing tools. As already stated in the last section, a static worst-case execution time analyzer requires both high-level information from the source code and low-level information gained from the machine code or

eventually the low-level intermediate representation. This data is available at the compilation time and is used intensively by the compiler, assembler, linker and loader but is not revealed to any other external tools. Thus, timing analyses have to reinvent the wheel by reconstructing this information once again. The wish list presents requirements from the WCET analysis community with the hope to influence compiler vendors to make this data available in the form of an intermediate format.

There have been some attempts to incorporate the worst-case execution time in a commercial compiler. [Bör96] designed a *WCET module* which in the first phase communicates with the frontend of a C-compiler for an Intel 8051 microcontroller in order to receive information on the control flow of the program in the form of pragmas. They have to be added manually by the user in the source code before the analysis. In the second phase, the *WCET module* passes this data to the backend in order to steer the code generation. Due to lack of time and programming experiences, the analyzing tool could not be finished and results are not available. Furthermore, this approach is less promising since it is desired to have most of the work being carried out by the tools automatically, avoiding any tedious tasks like adding pragmas to the source code by the user. Additionally, the prerequisite of user annotations (pragmas) is unrealistic since it assumes that the user has a thorough knowledge of the behavior of the program in any possible state.

[Eng98] addressed a similar problem considering the difficulties which come up when the source code is compiled by an optimizing compiler. The binding between the program source code and the object code becomes vague or even impossible to reestablish. To keep track of this relationship, which is mandatory for the static timing analysis, a framework has been designed based on a tight collaboration between the compiler and the WCET analyzer. Like the work introduced above, this framework could not be completely finalized within the given time. Essentially, the designed data structures were not powerful enough to hold information on *everyday* source codes which were considered to be too complex. Another drawback of this framework is its inhomogeneity where the WCET analyer cooperates with a third-party compiler. A more efficient solution is the use of a compiler which was designed in the same fashion as the framework and implemented by one team giving the opportunity of a tailored collaboration.

[LLPM04] proposed an approach to reduce the worst-case execution time at the expense of the code size and vice versa. This trade-off is deployed on a dual instruction set processor which supports both a reduced instruction set comprising 16-bit instructions and a full instruction set consisting of 32-bit instructions. Compiling a program completely with 16-bit instructions yields a program with minimum code size but also with the longest execution time. In contrast, the full instruction set counterpart results in a minimum execution time but a substantial increase in code size. The reason behind this controversial behavior is that a program using the full instruction set consists of fewer instructions, since a 32-bit instruction executes more operations than a 16-bit instruction.

The framework initially compiles the complete program using exclusively 16-bit instructions. The goal of the next step is to determine the set of blocks yielding the minimal

WCET. For this purpose, all basic blocks are experimentally translated into 32-bit instructions, and the emerging mode switch instructions are taken into account. Finally, the results are deployed and lead to a machine code with a mixed instruction set. To obtain tight WCET bounds, a timing analyzer was implemented assuming simplified conditions, namely the absence of cache memory and a simple pipeline structure of the processor. It could be shown that the worst-case execution time of a program can be effectively reduced by producing a relatively small increase in the code size.

The task pursued in [Byh04] is related to the present master's thesis in terms of the usage of the same WCET analyzer, *AbsInt's aiT Worst-Case Execution Time Analyzer*. Her studies aim at improving the determination of the WCET for real-time applications by integrating the static analysis tool into a development environment. The timing analyses could be successfully performed, but due to shortage of time and insufficient knowledge on measurements (dynamic analysis), the estimates could not be verified.

There are still some more efforts to incorporate a WCET analyzer into a development environment. But the author of this thesis is not aware of any successful result of a homogeneous framework where both the compiler and the timing analysis tool are tailored to the needs of its counterpart and permit an efficient communication.

1.5 Goals

The survey of the related works from the previous chapter points out that there is a lack of a homogeneous compiler framework with an integrated WCET analyzer. To accomplish this goal, both a thorough knowledge in compiler construction and an expertise in the subject of timing analysis is required. However, this combination is uncommon due to the fact that most software developers and research groups merely maintain their focus on one of the two issues.

The Computer Science Chair XII (Embedded Systems) at the University Dortmund has concentrated it research in the domain of compilers and embedded systems for many years and possesses a vast knowledge in this field. On the other hand, the company AbsInt Angewandte Informatik GmbH located in Saarbrücken, Germany, is a leading software developer of tools for validation, verification and certification of safety-critical software. One of their award-winning products is the aiT WCET Analyzer which statically analyzes cache and pipeline behavior to generate safe and tight worst-case execution bounds. Combining the skills of both teams yields a good starting point to create a WCET-aware compiler.

The compiler framework developed at Chair XII comprises an *intermediate representation* called the *Low-Level Intermediate Representation* (short *LLIR*). Its counterpart, the low-level intermediate representation used by AbsInt's WCET analyzer is the *CRL2* which is a mnemonic for *Control flow Representation Language*. The main objective of this the-

sis is the enrichment of the LLIR by WCET information. To meet this requirement, a conversion between both intermediate representations has to be established.

In detail, the goals pursued in this master's thesis are:

- **Integration of the WCET analyzer into a homogeneous compiler framework**
  The most efficient collaboration between a compiler and a WCET analyzer is accomplished when both modules are part of one homogeneous framework. Therefore, a seamless integration of AbsInt's timing analyzer is mandatory as well as a conversion between the LLIR and the CRL2.
- **Analysis of results generated by aiT**
  After passing the created control-flow graph in the form of a CRL2 file to aiT, the generated analyzer results have to be evaluated.
- **Extensions to the LLIR**
  The gained analyzer results must be transformed to the LLIR. For this purpose, a generic interface will be developed. It should be versatile, easy to use and not modify the existing LLIR code too extensively.
- **Realization of the mutual collaboration**
  To enable a mutual information exchange between both intermediate representations, a converter (LLIR to CRL2) as well as its counterpart have to be developed. Furthermore, the invocation of the analyzer toolchain must be preformed, and finally a generic mechanism to extend the LLIR by the extracted analyzer results must be developed.

The achieved results serve as a basis for future work concerning a WCET-aware compiler. In contrast to the state of the practice of using a compiler and an external WCET analyzer, the main advantage of this homogeneous framework is that the machine code is not the only interface between the WCET tool and the compiler. A compiler performs numerous analyses and gains a deep knowledge of the structure of the program which can be passed to the WCET analyzer resulting in more precise timing estimates. The extracted WCET information offers the opportunity to develop novel algorithms with the objective of optimizing the worst-case execution time of a program.

1.6 Outline of the Thesis

The thesis is organized in the following way:

- **Chapter 2** introduces the aiT WCET Analyzer. Its in- and output data as well as its workflow are briefly described.
- **Chapter 3** presents the structure of CRL2, the intermediate format used by the aiT toolchain to represent a program

- **Chapter 4** contains a description of the structure of the low-level intermediate representation (LLIR) frequently used at Chair XII, including its supported analyses and optimizations.
- **Chapter 5** provides a thorough overview of the requirements in order to achieve a WCET-annotated LLIR. Besides the conversion process between the two intermediate representations, a novel generic interface is presented. It employs versatile objectives as information carriers which can be simply interlinked with the LLIR by so-called handlers.
- **Chapter 6** describes the existing toolchain and presents the experimental evaluation to verify correctness of the extracted WCET estimates.
- **Chapter 7** concludes this thesis and provides an outlook on future work.

Chapter 2

aiT – AbsInt's WCET Analyzer

Nowadays, there are only few vendors providing commercial WCET analyzers. Also, research groups who focus on worst-case execution time analyses could not achieve any remarkable results. The few available products differ by supporting different processors, employing various calculation methods, as described in section 1.3, and cooperating with different compilers.

One of the leading analyzers is the aiT Worst-Case Execution Analyzer developed by the company AbsInt Angewandte Information GmbH [Abs05]. As the analyzer name suggests, it is a tool to determine the WCET of a program by static analyses. Thus, the results are valid for all inputs and each state of the program. In contrast to numerous other WCET analyzers, AbsInt's tool performs an analysis of the cache and pipeline behavior guaranteeing safe and tight timing bounds. To calculate the worst-case execution time bounds, it uses the *Implicit Path Enumeration Technique (IPET)* combined with *Integer Linear Programming (ILP)*. At the time this thesis was written, aiT was available for various processors, namely ARM7, HCS12/STAR12, PowerPC 555, PowerPC 565, ColdFire 5307, TMS320C33, C166/ST10 and the Infineon TriCore DSP v1.3. The latter is the processor this thesis is based upon. Although some parts of the software are tailored to this processor, the entire framework presented in this thesis can be easily ported to a different target architecture, since most of the code is processor independent.

2.1 In- and Outputs

There are two alternatives to analyzing a program using aiT. The most common way is to provide a binary executable of the program to be analyzed as input. To do so, the most convenient manner is to use the graphical user interface which performs the required execution of the toolchain in the background. It allows one to define the executable, specification files and further options that adjust the output of the result according to user's preferences. The specification file will be covered in section 2.2. There are several anal-

yses which can be selected in the GUI to be performed on the executable. The user can compute the call graph and the control-flow graph and have them both displayed in an external graph browser. Furthermore, the WCET analysis can be run and either pipeline states might be visualized, or the results can be merged into the graphs and displayed in an external graph viewer [ait].

A more advanced approach, which is also pursued in this master's thesis, is to invoke each program of the toolchain manually. This requires a thorough understanding of each tool, especially knowing the parameters which have to be passed on the command line for the desired operation.

The input file is assumed to be a binary executable which was generated from a restricted subset of ANSI-C without dynamic data structures or `setjmp` / `longjmp` statements. Besides the already mentioned specification files, the start address has to be set defining whether the entire program or just a snippet of the code will be analyzed.

Assuming that the program was run in an isolated environment without any external interferences, the output data is an upper WCET bound specified in cycles. When the clock rate is defined, the bound is additionally given in real time units e.g. seconds or milliseconds. Moreover, the results might be viewed in an external tool indicating various information like the worst-case execution path or the machine state at a particular point of the program.

2.2 Annotations and Specifications

To get correct timing bounds, aiT has to be provided with hardware specifications and user annotations, which can either be inserted directly into the source code or defined in a specification file (called *AIS file*). Some are mandatory to enable the analysis, others improve the precision of the results. However, this information should be carefully chosen since it is not checked by aiT and can lead to incorrect results.

Exemplary specifications concerning to the underlying hardware are:

- **Clock rate**
  The clock specification is essential to obtain WCET bounds in real-time units. It can be either defined as a single value or as a range restricted by a minimum and a maximum. Missing specifications result in bounds defined by cycles but with no information on the real time.
- **Memory areas and busses**
  To get precise results, the external memory areas with their minimum and maximum access times should be specified. Furthermore, they can be defined as read-only, write-only or containing either data or code.

- **Compiler name**
  The name is relevant for *exec2crl*, which is the first tool of the toolchain run by *aiT* and which is responsible for creating the intermediate representation *CRL2* (see chapter 3) from a binary executable. Providing information on the compiler used to generate the input file improves the understanding of the structure of the program and yields a more precise representation of the program.

Besides the hardware specifications, various user annotations may be defined:

- **Loop bounds**
  Although aiT tries to determine the number of loop iterations automatically using a *loop bound analyzer*, this succeeds only for simply structured loops with constant bounds. These loops consist of machine code matching particular patterns which are expected to be generated by the supported compiler. The remaining loop bounds have to be defined manually.
- **Targets of computed calls and branches**
  To get a proper reconstruction of the control flow from a binary file, the target addresses of computed calls and branches have to be available. Usually, they are found automatically, but for some non-trivial calls and branches, the addresses can not be resolved. These targets have to be specified manually.
- **Recursion depth**
  It is mandatory to define upper bounds for the recursion depths of all recursive routines. Missing depths prevent the WCET analyzer from progressing.
- **Register values**
  The value analysis, as part of the toolchain, tries to determine register values for every program point. In cases of failure, the values can be specified manually.

The aforementioned specifications are by no means complete. The full list can be found in [ait].

2.3 Workflow of WCET analyses

The graphical user interface as well as the manual approach have to cope with the appropriate invocation of each tool representing a portion of the aiT toolchain. The analysis is performed in six main steps, including the reconstruction of the control-flow graph, the value analysis, the loop bound analysis, the cache and pipeline analysis and the path analysis [Fer04]. The structure of the framework is depicted in figure 2.1.

Reconstruction of the control-flow graph
The ordinary way a WCET analysis is performed begins by supplying aiT with a bi-

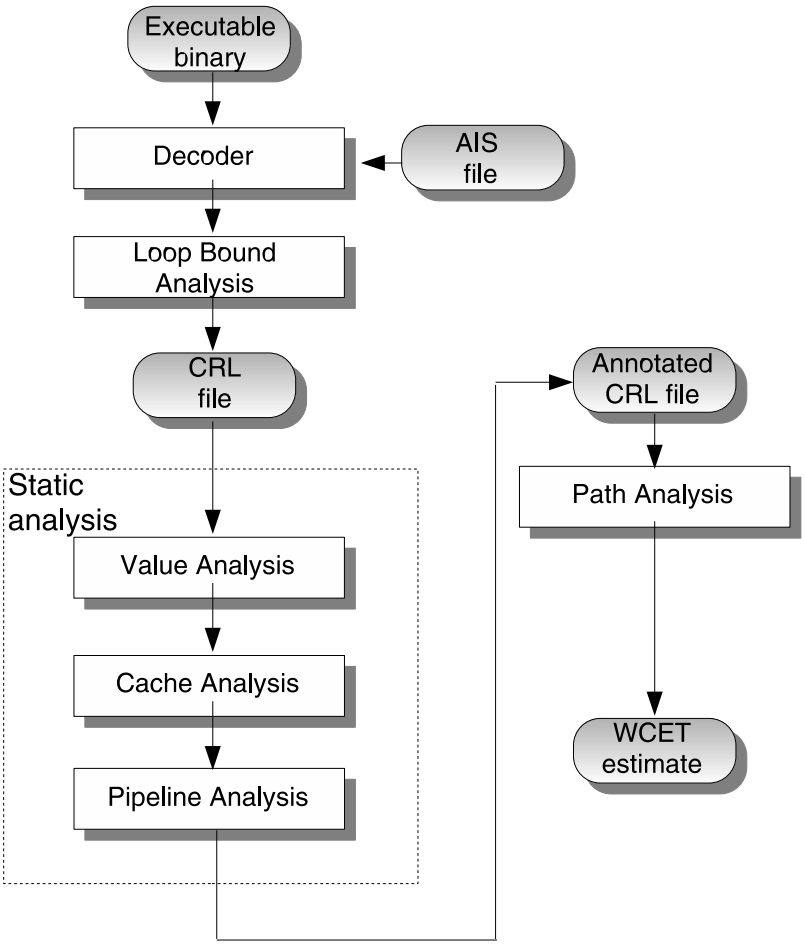


Figure 2.1: Analyses performed by the WCET analyzer aiT

nary executable. AbsInt's decoder (called *exec2crl*) reads the input file and reconstructs a control-flow graph (CFG). In doing so, exec2crl relies on some known patterns which are expected to be generated by a particular compiler, like for example specific instructions which represent branches and calls. Thus, the analysis framework uses specialized decoders which are customized to a particular compiler. Finally, the graph is annotated with data used in subsequent phases and is translated into AbsInt's intermediate format called CRL2.

One of the contributions of this thesis is the replacement of this reconstruction step by a novel converter. As already pointed out in section 1.5, a program represented by assembly or machine code serving as the only interface between the WCET analyzer and the compiler framework is a less efficient approach. The reason for that is that beneficial information available to the compiler is not handed to the analyzer. Within the framework presented in this thesis, all essential information required to generate the control-flow graph is read out from the low-level intermediate representation of the compiler framework which was created by the compiler in preceding phases. The remaining task to establish a sound

basis for further analysis steps is to translate the control flow data into CRL2. This work is carried out by the converter *llir2crl*.

Value Analysis
The value analysis investigates potential values in the processor registers for any possible program point and execution context. In some cases, explicit values are untraceable leading to a definition of lower and upper bounds. The analysis is based on the model of *abstract interpretation*, a theory of establishing approximations of the semantics of a program in a static fashion without the need of performing the overall calculations [CC77]. In more detail, aiT is mapping the input states of each machine instruction to a more abstract output state, resulting in a semantically equal but more suitable form of the program representation for further analyses. The results serve various purposes in the subsequent phases. The loop bound analysis expects register values in order to find upper bounds for the number of loop iterations. The cache analysis requires predicted values to designate possible accesses of data and indirect memory accesses. Last but not least, the predicted values contribute to determining infeasible paths that result from conditions being true or false at any point of the program.

Loop Bound Analysis
Apparently, the major part of the execution time of a program is spent in loops, making their iteration numbers an important issue. aiT relies on precise bounds to be able to perform a WCET analysis at all. The loop bound analysis tries to find predefined patterns for loops which are supposed to be generated by a certain compiler. Furthermore, it reverts to the results produced by the value analysis to determine the bounds for the iteration numbers. This procedure is repeated as long as all loops are handled and eventually new contexts are added. At the end, all relevant information is accumulated in order to perform the final stage of the value analysis. The detection of loops succeeds only for simple structures and demands manual specifications of the iteration bounds in the form of user annotations for the remaining loops. Both the value and the loop bound analysis are performed by the tool *tricoredaan*.

Cache Analysis
The execution of this analysis step is optional and depends on the processor architecture of the underlying hardware. Absence of cache memory makes the cache analysis redundant, otherwise aiT statically analyzes the cache behavior of the program using a formal cache model. Accesses to main memory are examined by an algorithm distinguishing among *sure cache hits* and *unclassified accesses*. The proper analysis relies on the value ranges of processor registers obtained from the value analysis. Like the value analysis, this approach is based on the framework of abstract interpretion.

Pipeline Analysis
The last static analysis is the pipeline analysis which models the pipeline behavior. It is based on the current state of the pipeline, the resources in use, the contents of prefetched queues and the results received from the cache analysis. It aims at finding a WCET esti-

mate for each basic block, which is assigned to the corresponding outgoing control flow edge that relies on the context. Involving the theory of abstract interpretation, each basic block is analyzed by taking the results of the pipeline states obtained from the preceding basic block into account and by incrementally proceed with succeeding machine instructions. In the last iteration step, the final results are passed on to the next basic block. Finally, the maximum of simulated cycles for each flow of instructions of a basic block is determined; it designates the longest time the basic block takes to execute. The pipeline analysis is managed by the application *tricorepipe*.

Path Analysis
The path analysis is based on the Implicit Path Enumeration Technique (see section 1.3). For each context c and each edge e, the execution count $C(e,c)$ is determined. Additionally, the WCET estimates generated by the pipeline analysis are taken into account. They are expressed by $T(e,c)$, being the worst-case execution time for an edge e and a context c. Thus, a specific path in a given context c including a limited set of edges e yields the execution time which results from the sum of $C(e,c) \cdot W(e,c)$ over all edges. The objective of the path analyzer *pathan2* is to determine the global WCET by maximizing the product of $C(e,c)$ and $W(e,c)$ over all edge-context pairs for a feasible path. The condition of a feasible path imposes constraints, which are accumulated to a linear constraint system, and which are solved by *integer linear programming* with the support of the auxiliary tool *lp_solve*. Basically, the conditions are derived from the structure of the program, the loop bound analysis and user annotations specified in the AIS file and the source code.

The full run of the WCET analyzer comprises twelve sub-steps in total. Besides the main six steps mentioned above, the remaining steps are outside of scope of this thesis and therefore not considered in detail. For example, aiT offers the opportunity to view the control-flow graph combined with the calculated results in an external graph browser, AbsInt's *aiSee*.

Chapter 3

CRL2 – AbsInt's Low Level IR

Every analyzer and code generator operating on a program requires an intermediate representation in order to apply analysis and optimization algorithms. Depending on the demands, different levels of representation are employed. *High-level intermediate representations* directly correspond to the syntactic structure of the source language and are used in the earliest stage, often in the form of an *abstract syntax tree*. A more abstract format is the *medium-level intermediate representation* that uses a language-independent notation. In the compiler domain, a widespread optimization used at this level is the *common subexpression elimination*. The *low-level intermediate representation*, as the third class, is highly dependent on the processor architecture. The format is often based on machine instructions and uses assembly mnemonics.

aiT applies its decoder to extract a safe and precise control flow from a binary executable, and requires a low-level intermediate representation to hold the control flow graph. *CRL2 (Control Flow Representation Language, 2nd version)* is a generic library used for this purpose. It is a human-readable intermediate format representing a control-flow graph for static analysis. Initially it was developed at the University of Saarland [Lan98] and is now maintained by the company AbsInt. The format was designed with the objective of providing a simplified interface for analysis and optimization algorithms at the assembly level. Also, the structure of CRL2 is designed in a manner to support various hardware architectures.

3.1 Key Components of CRL2

The control-flow graph is described by syntax elements, called *items*. The main item is the CRL2 graph. It holds both the control-flow graph and the call graph, deploying various other CRL2 components [CRL05]. Figure 3.1 illustrates the hierarchical structure of the CRL2 library.

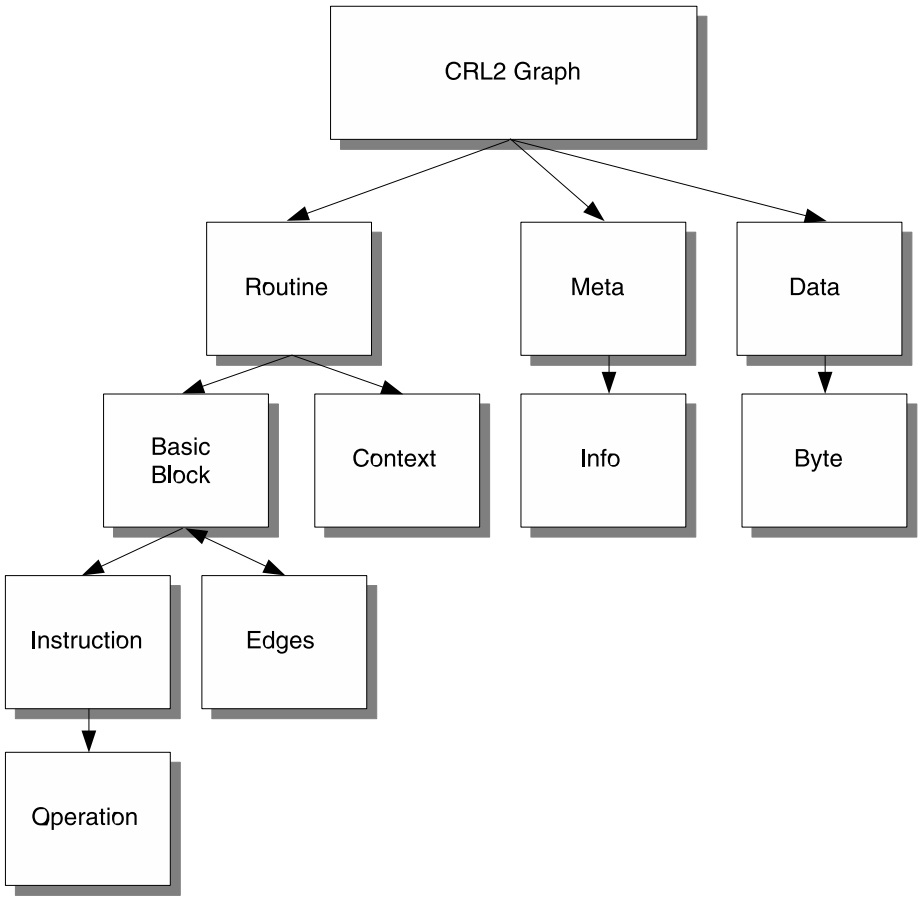


Figure 3.1: Structure of the CRL2 intermediate representation

The *CRL2 graph* contains the global structures, blocks of data and meta information. The *data* items store *byte* items which represent constant data used in the binary. These data sections are extracted by the value analysis in order to analyze memory accesses. Entities of the *Meta* class hold *info* items which accommodate symbol tables and complex user annotations.

CRL2 routines, as the next component of the hierarchy, represent a sequence of basic blocks and may have different origins. Like assembly procedures, the binary executable and the resultant CRL2 intermediate representation are partitioned into routines which are derived from ordinary functions of the source code. Although there are no strict conventions for the routine names chosen by the compiler, the state of the practice is to label them as a concatenated string of the function name and a preceding underscore. Another origin of CRL2 routines results from the fact that the intermediate representation is generated from a file which was produced by the linker. To generate a binary executable which can be interpreted by a machine, the linker merges all required source files as well as libraries and adds specific start routines (called *startup code*) such as '__main' [Lev99]. Consequently, these auxiliary routines are read by AbsInt's decoder *exec2crl* and are trans-

formed into CRL2 routines. In addition to some special functions, which are spawned by hand-written assembly routines, several routines result from the *loop transformation*. This process, which consists of a repetitive invocation of the loop bound and value analysis, modifies the original control-flow graph by splitting up the loops into separate routines. These routines call themselves recursively. The goal of the loop transformation is to improve the WCET estimates by creating additional contexts. This approach will be explained in more detail in section 3.2.

Basic blocks, which are implemented as *CRL2 blocks*, form the nodes of a control-flow graph. To complete the graph structure and allow to model execution paths of a program, basic blocks are joined by *CRL2 edges*. Depending on the represented jump in the control flow, the edges are categorized in various types. For instance, incoming and outgoing edges are deployed to reflect loops, or the edge pair of call and return edges, which simulates the proper program flow after a call instruction.

Like in the most intermediate representations, blocks consist of a sequence of *CRL2 instructions* which in turn are composed of *CRL2 operations*. Depending on the processor architecture, an instruction might refer to a single operation or a set of processor operations which will be executed in parallel on e.g. a VLIW processor. For paged architectures, instructions can be assigned both the base and the page address. This combined form designates the bus address of the instruction, before it is translated by the memory management unit (MMU).

The elementary unit, the operation, is designated by a unique numerical identification called the opcode. Furthermore, it accommodates a mnemonic, which is a human-readable representation, and additionally a list of operands used in this operation. The operands can be register names, constants or symbols. They are classified into source and destination resources. *Source resources* represent resources of the operations which are read, namely registers, constant values or abstract resources, like memory or stack (called *symbols*). On the other hand, the vector of *destination resources* denotes resources which are written by the operation, this is registers and symbols. Due to the nature of constants, they can not be altered and are therefore not be considered as writable resources. In the CRL2 format, the structure of complex operations used with various processors can be illustrated by a hierarchical representation. Notably in CISC architectures, some operations consist of complex operands that are executed in a specific order. This extensive structure is modeled by a hierarchy that is called an extension. *Extensions* are assigned to an operand and hold, like other CRL2 items, various details. Besides the information on the order, each operand is treated akin to an operation. It is denoted by an unique opcode and contains its own lists of involved resources that again are classified in *source* and *destination resources*. Moreover, further extensions can be added to operands that are already part of a superior extension. In order to generate a proper representation of a CRL2 operation with all mandatory declarations, the TF14NET framework and a particular specification file, which is provided for each processor architecture, are employed. The latter, called a *NET file*, holds all information on each operation specified in the instruction set architecture

(ISA) of a particular processor. The specification file is based upon the corresponding reference manual and holds information on valid operands, addressing modes, cycles used for the execution, latencies and the hierarchical structure represented by the extensions. To create a CRL2 instruction, the TF14NET library is supplied with the involved operands and the unique opcode. Based on this input data, the library reverts to the NET file, parses it and finally calculates a fully structured operation wrapped in a CRL2 instruction.

Within the CRL2 framework, each item stores its characteristics in *attributes*. They are used to accommodate all data required to represent a program on a low level. Furthermore, all results gained during the analysis process are stored in attributes. They serve as an interface between individual analysis steps by exchanging the essential information. For instance, the pipeline analysis calculates the worst-case execution time for each basic block and archives these values in the context-dependent attribute called *pipe_cycles*. Subsequently, these results are used by the path analysis, in order to calculate the global WCET.

3.2 Contexts

The issue of *contexts* plays a prominent role in the *data-flow analysis (DFA)*. The latter operates on the abstract representation of a program given by a control-flow graph. The task of the DFA is to determine an objective function for each node of the graph which maps any possible input data to a proper output. To do so, a system of *data-flow equations* for each node is set up, expressing the relationship of the incoming and outgoing information. In the final step, the system has to be solved.

The results from the data-flow analysis provide the basis for the static program analysis. The abstract data-flow estimations for each node must not violate the actual state of the program at any point. However, this condition might lead to oversized value ranges, which in turn produce imprecise timing estimations, mainly for the WCET and BCET. A solution to this dilemma is the use of contexts.

In general, contexts narrow the vast value ranges by splitting them up into smaller sets, reflecting a more subtle view of the current state of the program. This allows to perform multiple context-dependent analyses, which result in more precise timing estimations. Contexts are assigned to CRL2 routines and are expressed by a string, the call string. *Call strings* are comprised of previously created call strings, identifying the sequence of calls, beginning with the entry point of the analysis. They are of the form "$B \longrightarrow R$", where B is the basic block containing the call instruction to the routine R.

As already mentioned in the previous section, CRL2 routines may be derived from source code functions. Due to the nature of functions, they have to cope with arguments which are passed during the call. When contexts are not taken into account, the abstract data-flow information must satisfy the large value range of all potential actual arguments.

Notably, the value analysis suffers from this fuzzy specification. The investigation of possible values in the processor registers is limited to some bounds which hardly state the realistic behavior of the program. Using contexts can substantially improve the results generated by the value analysis. Assume, a routine is called twice with arguments 0 and 10, respectively. Without contexts, the interval $[0, 10]$ designates the abstract specification for the register r that stores the values of the routine argument. Thus, it is implied that r might accommodate any value between 0 and 10, leading to imprecise conclusions. A more efficient approach is to use two contexts, which define the corresponding intervals as $[0, 0]$ and $[10, 10]$, respectively. This way, the value analysis is provided with precise input omitting the redundant values of the interval $[1, 9]$.

Furthermore, the adoption of contexts is beneficial for the path analysis. Considering various contexts admits to find parts of the code which are never executed at a specific point of the program. Assume the following C function (in a simplified view):

```
int foo( int x ) {

  if ( x != 10 ) {
    ...  // takes 100 cycles
  } else {
    ... // takes 5 cycles
  }

}
```

The function `foo` contains an argument `x` which affects the control flow within the function body. If `x` is not equal to `10`, then the following block takes 100 cycles to execute. Otherwise, the **`else`**-block with 5 cycles of execution time is traversed. Assume, `foo` is called twice, with argument `10` for the first call and with argument `y`, which can not be statically specified, in the second call. Regarding the less sophisticated scenario without contexts, the worst case must be presumed in order to guarantee the safety of the WCET estimates. This means that the worst-case execution time of the function `foo` always constitutes at least 100 cycles, independently of the value of the function argument `x`. The performance of the path analysis can be drastically increased by imposing contexts for the distinct function calls. Context c_0 denotes the call `foo(10)`, the second context c_1 the function call `foo(y)`. The analysis, which traverses the control-flow graph, has the ability to distinguish among different paths. Taking the first context c_0 into account, the function `foo`, omitting the `then`-block of the **`if`**-statement, contributes 5 cycles to the accumulated WCET. For the second context c_1, where `y` is still unknown, the analysis carries on applying the worst case which enlarges the current WCET by 100 cycles. As a conclusion, the introduction of contexts possibly means a reduction of the accumulated WCET by 95 cycles. In present-day real-time applications which frequently encompass several thousands lines of code, the neglect of contexts yields a notable overestimate of the timing information.

The proper handling of loops is of great importance since programs spend most of their runtime in loop bodies. Like for routines, the *loop transformation*, which is an inherent part of the aiT WCET analyzer, exploits the benefits of contexts in terms of loops. Due to the processor cache, the execution time of the first loop iterations may vary. When the loop is executed for the very first time, the cache is consulted and often found not to contain any data of the loop. This situation is known as a cache miss and necessitates fetching the information from memory. In the second iteration, it might happen that the loop data in the cache is still altered. The reason for that are mechanisms performed by sophisticated cache replacement algorithms, which are known as *speculative prefetching*. If the loop iteration continues, the cache situation stabilizes and involves a direct data access from the cache (called cache hits). To handle the individual conditions, the loops are *virtually unrolled* by the loop transformation, which is based on the *VIVU approach* [MAWF98]. With the goal of enhancing the WCET estimates, aiT has to examine loops in a specific fashion. Since contexts are exclusively assigned to CRL2 routines, loops are compelled to be extracted. More accurate, basic blocks representing a loop are transformed from the original routine into a newly created one. Thus, various contexts, reflecting the cache behavior, can be appended. This distinction of contexts entails a more precise estimate, since context-dependent paths with distinct execution times may be taken into account. A thorough description of the loop transformation is given in [ait]. Last but not least, it should be mentioned that the loop transformation operates solely on an internal intermediate representation of the binary executable, i.e. all modifications remain in the CRL2 data structure and never affect the executable program by itself.

There are two sides of the coin when it comes to the increasing number of contexts. As previously mentioned, the WCET estimates are positively affected by a more subtle distinction of contexts. However, on the other hand, an enormous number of contexts demands a longer analysis time. Programs with a complex calling structure or nontrivial nested loops induce a high increase of contexts, and may disable an efficient timing analysis. Therefore, a trade-off between computation time and precision might be inevitable for larger programs. aiT offers the opportunity to restrict the number of created contexts by limiting the length of the call string. This is done by adding the option *interproc max-length = n* to the AIS file, with *n* being the length of the call string. To control the number of loop contexts created by the loop transformation, the parameter *interproc unroll = n* may be appended to the AIS file. Furthermore, the parameters *flexible and limited* steer the loop detection. The first parameter relies on the user annotations and the automatic loop bound detection, whereas the latter coerces the analyzer into merely interpreting the manually defined loop bounds, which, under these circumstances, must be given for every loop.

Chapter 4

Existing Low Level IR (LLIR)

The Low-Level Intermediate Representation (LLIR) is an intermediate representation which offers the opportunity to accommodate assembly code in the individual stages of a compiler backend. The assembly code can either be received from a code selector or might be directly imported from an assembly file. The elaborate structure of the LLIR allows the modeling of dependencies among the elements which results in a control-flow graph. Based on this representation, various modifications can be performed. Furthermore, the LLIR library provides analyses and optimizations enabling the generation of optimized code within a compiler framework. In addition to the import, the library provides functions which export the modified assembly code into an assembly file. The overview of the elementary sets of the LLIR (I/O functions, internal data base and library of analyses and optimizations) is given in figure 4.1.

The software was developed by the Informatik Centrum Dortmund (ICD) [ICD05] and has been successfully employed for various research projects at the Embedded Systems Groups of the Computer Science Department at Dortmund University. For instance, it was deployed in the design flow of an application specific processor (ASIP) for network protocol processing [NW04]. Moreover, the LLIR was involved for different issues, which were subject of several master's theses, e.g. for bit-level optimizations [Pyk03]. This intermediate representation is also the objective low-level intermediate representation which is used for the present master's thesis, as a counterpart to AbsInt's CRL2.

4.1 Key Components of LLIR

Like most low intermediate representations, the LLIR is comprised of the key components such as functions (equivalent to routines), basic blocks, instructions and operations. Due to the analog structure of the CRL2, a transformation of control-flow graphs, involving some minor adjustments, is feasible. The main difference among both representations is the reconstruction of the CFG. The LLIR gains its data either from an assembly file or

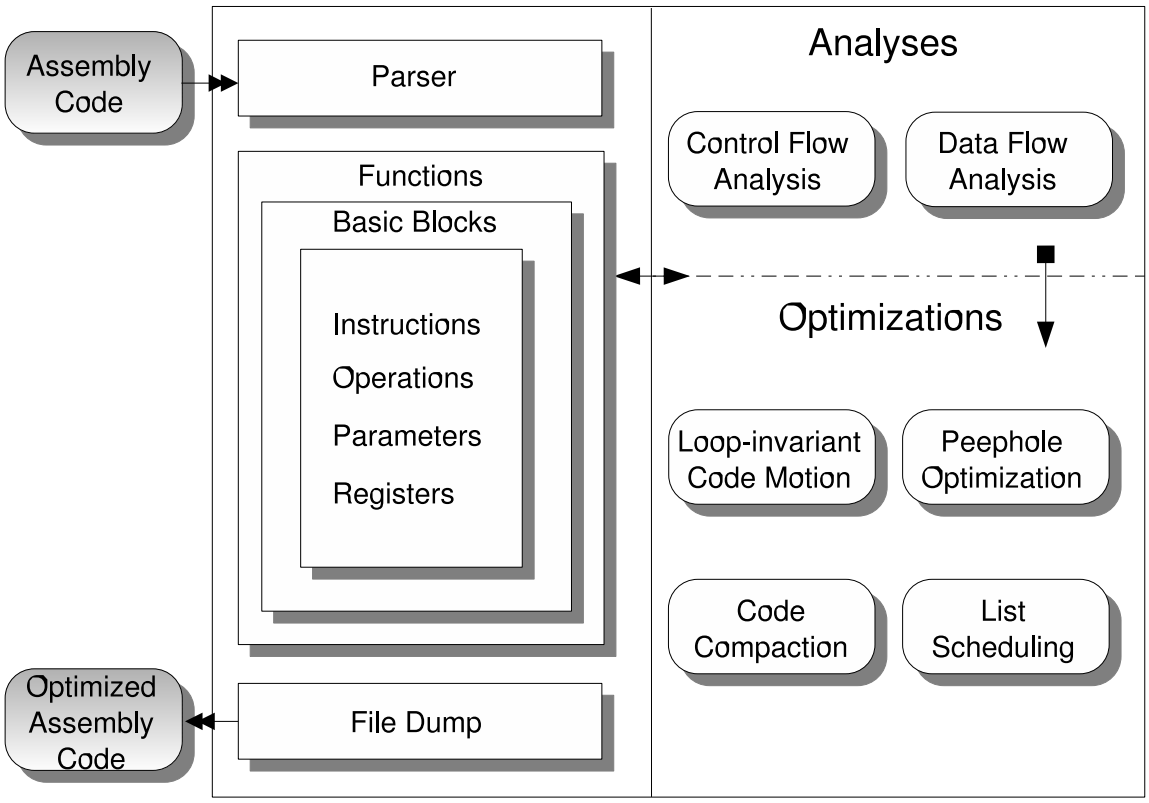


Figure 4.1: Structural overview of the Low-Level Intermediate Representation (LLIR)

obtains it from an intermediate representation which is operating on a higher level. In contrast, the CRL2 library relies on data which was extracted from a binary executable. The program file is converted into a readable format by using a disassembler. Thus, more concrete information on the program is available, e.g. additional functions which were used by the linker. Despite the absence of these specific details, the knowledge of the program, which is accommodated in the LLIR, satisfies to serve as input for the aiT WCET analyzer.

The general design of the LLIR is depicted in figure 4.2. The main class is the LLIR which represents the entire assembly code. Due to the fact that further key components in the structure hierarchy are positioned beneath this class, it offers a global view. Thus, most analyses and optimizations, which will be described in section 4.2, are performed at this level.

The subsequent class, the *LLIR_Function*, represents assembly procedures which are ordinarily derived from source code functions. It is composed of a sequence of basic blocks and manages their dependencies in the form of a control-flow graph. Basic blocks are represented by the class *LLIR_BB* which store straight-line code. They are designated by a unique label name and contain a set of instructions realized by the class *LLIR_Instruction*. In addition, they meet the constraint imposed by the CFG theory which grants jumps to

be performed exclusively by the last instruction. Instructions hold one or more machine operations which are executed in parallel. Depending on the processor architecture, an instruction is assigned exactly one operation (most systems) or, as for VLIW architectures, multiple operations. Machine operations, realized by the *LLIR_Operation* class, are specified by a mnemonic and a list of involved resources, called parameters.

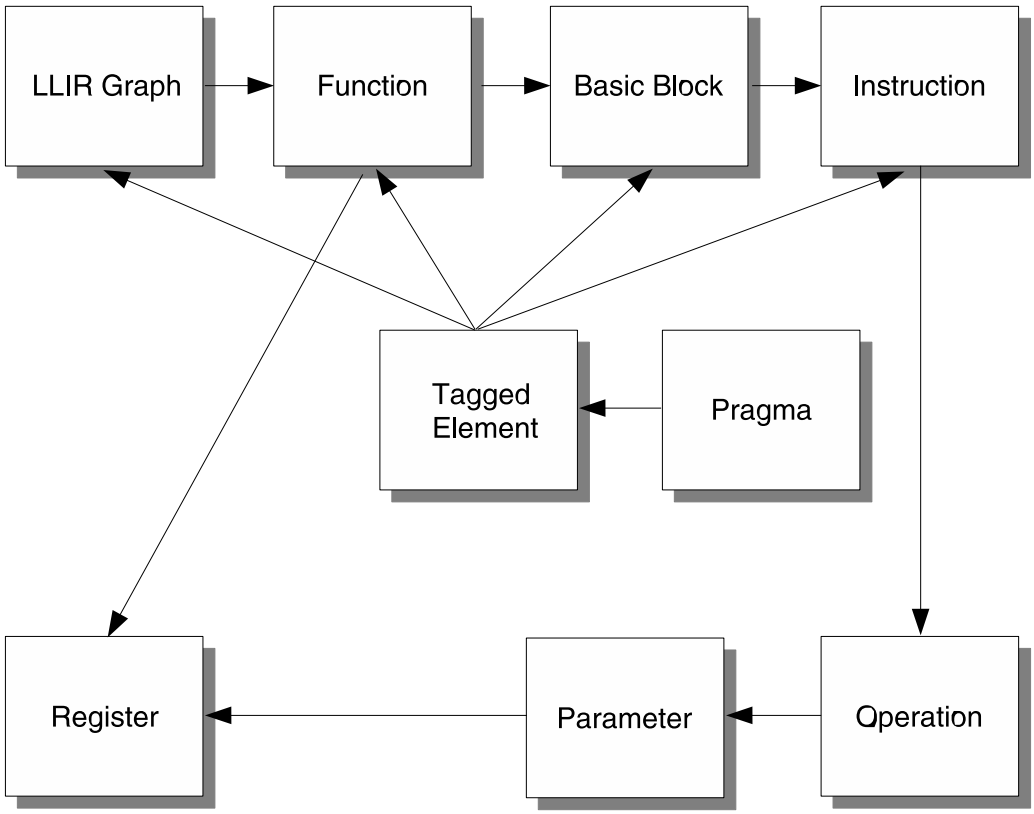


Figure 4.2: Class diagram of key elements of the LLIR

The class *LLIR_Parameter* distinguishes among four different types, namely constants, labels, operators and registers. Additionally, parameters are characterized by their usage; those resources which are written are denoted by *DEF*, those which are merely read are called *USE*. Finally, parameters, which are both read and written, are designated by *DEFUSE*. *Constants* represent standard integer constant values and, due to their nature, are invariably of type USE. The other two elementary parameter types are *labels* and *operators* which denote jump labels and architecture specific tokens, respectively. *Registers*, as the last parameter type, are more complex and are modeled by the class *LLIR_Register*. They are managed at the function level, i.e. each LLIR function contains references to all registers that are involved in the underlying LLIR elements. This allows the comparison of registers in related objects by a simple pointer comparison. Each register is identified by a unique name and can be either a physical register of the underlying hardware or a virtual register which is afterwards assigned a physical register name by the register allocator. Furthermore, registers can be assembled into complex register hierarchies, e.g. for

the Infineon TriCore DSP which supports extended 64-bit registers that are a compound of two 32-bit data registers.

In addition to the program structure, the LLIR stores further details on the assembly program. The information is either a simple string or an index to an enumeration of predefined strings which is represented by the class *LLIR_Pragma*. The pragmas are summarized hierarchically into the class *LLIR_TaggedElement* which serves as a base class for all other classes expect LLIR_Parameter and LLIR_Register. Furthermore, basic blocks, instructions, operations and pragmas might be tagged with information concerning their location in the input file, that is the name of the assembly file and the corresponding line number.

The Low-Level Intermediate Representation was designed with the focus on portability. In order to adapt the library to a specific target architecture, a processor specification file has to be modified. The mandatory steps concerning the specification file are described in more detail in the following. The register set of the underlying target architecture is to be defined. Besides the register names for the physical resources, internal registers, virtual registers and the register hierarchy are to be specified. *Internal registers* are auxiliary hardware resources which do not occur in the assembly file but might be helpful for internal operations. *Virtual registers*, also called pseudo-registers, are auxiliary resources, which are utilized by the code generator leading to a more flexible handling. For example, the code selector emits code without taking care of the limited number of available physical registers, by simply assuming that an unlimited amount of virtual registers is available. In the subsequent step, the register allocator takes care of multiplexing the large number of virtual registers onto a limited number of processor registers. Last but not least, supported register hierarchies might be modeled. To do so, extended register names and the registers involved in this relationship must be specified. Thus, complex registers can be either addressed as a whole or through their children. Another significant adaption is the definition of the instruction set provided by the target architecture. Besides the enumeration of each instruction, a cost table has to be defined. It indicates the cost of each instruction, usually corresponding to the individual memory requirements, and can be used within a tree pattern matcher of the code selector. The remaining adoptions concern the operators and predefined pragmas. *Operators* are special tokens which are used as operation parameters for various purposes, e.g. to identify the used address mode. Application specific pragmas might be predefined to customize supplementary information which can be tagged to most of the LLIR components.

4.2 Analyses and Optimizations

Besides the possibility of transforming an assembly file into an object structure, the LLIR provides mechanisms for analysis as well as for optimizations. The analyses proceed transparently for the user, i.e. any modifications concerning an LLIR component, which affect

the control-flow graph, are recognized automatically. The analysis results are then instantly updated. The latter are mainly retrieved by the optimization algorithms which are notably involved in the final stage of the backend. There are two typical process flows. In a homogeneous compiler framework, the program is passed from the code selector to the backend, where it is optimized and finally stored as an optimized assembly code. The latter is supplied to the linker. Another application which implies that the assembly code is already given (by an independent code selector or written by hand), is the *post-pass optimization* [NW04]. The assembly file is transformed into the LLIR, optimized and dumped out in the form of a more efficient assembly code. The following sections briefly introduce the supported analyses and optimization techniques.

Control-Flow Analysis

In order to traverse the control-flow graph, the class LLIR_BB provides data structures and functions to express dependencies in the form of incoming and outgoing edges among basic blocks. There are three relationships between basic blocks which are managed [ICD01]: reachability, domination relation and loop depth. *Reachability* indicates whether the CFG contains a valid path between two given blocks and can be applied to detect dead code. *Domination relation* is a concept of one basic block always being before or after another given block on all execution paths. Finally, the loop depth represents the loop nest level of a basic block.

Lifetime Analysis

The lifetime results are essential for the data flow analysis. They concern the relation of definition and use points for a specific register [Mor98]. A register is called *live* on all paths from its definitions to its uses. This information is notably relevant for register allocators, in order to detect virtual registers which are no longer used and hence can be reallocated. The results are based on specifications for each register which point out the def/use properties before and after the execution of the involved instructions.

Def/use Chain Analysis

A related approach is the def/use chain analysis, being also part of the data flow analysis. Various optimization algorithms, as the value propagation optimizations, need to know the relationships between the read access of a register, designated as *USE*, and the point where it was written (known as *DEF*). The results are obtained by conducting the evaluation for each operation and taking even complex register hierarchies into account. To get a pictorial view, the user might even create a def/use chain graph.

Loop-invariant Code Motion

The first code optimization technique moves assembly instructions which are located within a loop, but are not affected by the loop statements before or after the loop body. In order to fully exploit this approach, it should be applied in both the compiler frontend and the compiler backend. The former is described in section 1.2. The objective of performing

the code motion at the low level is to shift supplementary assembly instructions which were added by the code selector as well as by the register allocator. Shifting the instructions outside of the loop results in a reduced number of executions which finally yields a speedup.

Peephole Optimization
Code generators often produce code that is not efficient but contains redundant instructions. The peephole optimization is a simple but effective approach to eliminate these redundant instructions. It improves the code locally by scanning short sequences of instructions and eventually substituting them by a shortened sequence. If the goal of the optimization is to improve execution time rather than code size, the considered sequence is replaced by a faster one which can cause an increase of the code size. The term *peephole* refers to the size of the window which is moved on the target program [ASU86]. Due to the fact that each improvement might produce new opportunities, the peephole optimizer is run manifold. To use this technique with the LLIR, the optimization pattern, the peephole size and further specifications must be supplied.

Code Compaction
This algorithm might be used to exploit parallelization of instructions. On the one hand, it can be applied with VLIW architectures to force the execution of several operations of one instruction in parallel. On the other hand, the assembly code of a target architecture with a single operation per instruction might be optimized in terms of parallelization, by replacing simple operations by a complex one. To do so, the LLIR is to be provided with distinct parameters, e.g. operations that spawn a conflict when used at the same time.

List Scheduling
This scheduling mechanism forms the final stage of the backend optimizations. It verifies whether all constraints concerning the mandatory latencies of instructions are met. The constraints are to be modeled for each target architecture. For instance, it can be defined that two instructions must have some delay cycles before being executed consecutively. Whenever the list scheduler discovers a violation of the latency constraints, it either adds the required number of NOP (No Operation) instructions or rearranges the order of instructions.

Chapter 5

Extensions to the LLIR

5.1 Introduction

The Low-Level Intermediate Representation (LLIR) was primarily designed to hold information extracted from an assembly file. The data exclusively represents the structural elements of a program that are used to construct a control-flow graph. The LLIR provides algorithms to analyze and optimize the accommodated program and thus extends the knowledge of the program structure. However, none of the auxiliary data concerns the timing behavior of the structure elements.

As stated in chapter 1, timing information is essential for embedded system applications. It might be used to prove the correctness of a given system. Furthermore, timing estimates like the worst-case execution time (WCET) must be known in order to design optimized hardware. The absence of this information can have a harming effect on the system in terms of both over- and underestimation. On the one hand, designers try to bypass this dilemma by using oversized target hardware. Needless to say, the waste of resources boosts the production costs and endangers the competitiveness. On the other hand, the hardware might not suffice to meet all timing constraints. Their violation prevents the system from a proper operation, ranging from a negligent degradation to a physical damage or even a catastrophe.

In order to cope with real-time systems, the LLIR needs to be extended. The library has to offer the opportunity to store supplementary information, notably those which concern timing behavior. This data is obtained from AbsInt's aiT WCET analyzer which was introduced in chapter 2. The analyzer is incorporated together with the LLIR into a homogeneous framework. The LLIR exports the program it represents into a CRL2 file and passes this to the WCET analyzer. Subsequently, aiT performs its WCET analysis and provides its results in the form of a fully annotated CRL2 file which is finally re-imported into the LLIR.

The main objective of the present thesis is to design and realize adequate concepts of

extensions to the LLIR in order to hold worst-case execution time estimates. These extensions enable the development of novel algorithms which can be employed in the domain of WCET compiler optimizations. To accomplish this goal, the existing LLIR has to be extended by elaborate data structures. Moreover, a generic interface has to be provided simplifying the access to the WCET information. It is meant to hide the complex inner access mechanisms from the user and design it as transparent as possible. The LLIR is supposed to provide simple `get` functions which are invoked by the user (or by the optimization algorithms) in such a manner that no redundant function arguments are required to be passed. For example, to receive the WCET of a specific basic block, the user merely invokes a universal `get` function of that block and specifies the additional information which indicates that WCET data is sought.

By explicitly specifying the type of information searched or manipulated within the LLIR, a flexible, generic and extendable interface is designed which is not restricted exclusively to WCET data. This generic interface allows the development of new compiler objectives - apart from WCET considered in this thesis - and to keep the simple access concepts. Thus, a maximum degree of extensibility and reusability is provided making the LLIR a convenient platform for distinct compiler optimizations. The remaining sections of this chapter thoroughly explain the concepts and realizations which are involved in the novel extensions to the LLIR concerning the WCET issue.

5.2 Conversion from LLIR to CRL2

The goal of the first phase is to supply the aiT WCET analyzer with a program which is supposed to be analyzed in terms of its worst-case execution time behavior. As input data, aiT demands a program in the form of its internal format, namely AbsInt's CRL2 intermediate representation. The ordinary usage envisions a binary executable as input file that is disassembled by AbsInt's decoder (called exec2crl). Based on this information, a control-flow graph and a call graph are constructed and stored in the CRL2 format. The latter is passed to further analysis tools of the aiT toolchain and progressively extended by the analysis results.

The general concern with external tools, which are integrated into a compiler framework, is the lack of available information. In the beginning, the compiler is supplied with a program which is written in a source language, mostly a high-level language like C or C++. This program is transformed into an intermediate representation and various analysis and optimization algorithms are applied which in turn construct and modify a control-flow graph (CFG). This complex data structure is annotated with individual attributes which reflect the results of the previous steps. Due to the fact that a lot of effort has gone into the creation of the CFG, this stage presents the optimal point of entry for the integration of an external analysis tool. To put it in other words, the most efficient way to analyze a program is to provide this gained information as input for the external WCET analyzer.

Using an inhomogeneous platform that contains the compiler framework on the one hand and a separate WCET analyzer on the other hand leads to a waste of information. The compiler generates a binary executable which contains merely the mandatory details of the program in order to be interpreted by the machine. Supplementary information that was gained during the compilation process is omitted. When this file is provided as the only interface between the compiler and the WCET analyzer, valuable data gets lost. In addition, the analyzer faces the chore of reconstructing the control-flow graph once again.

In contrast to the ordinary workflow, this thesis pursues a modified approach. To elude the problem mentioned above, the framework developed in this thesis combines both toolchains into one homogeneous system. In the first step, the source program is transformed into the LLIR. Based on this control flow information, the developed converter (called *llir2crl*) generates a CRL2 file that is used as input to the aiT WCET analyzer. This implies, that the original workflow of aiT is shortened. The process of constructing a CRL2 intermediate representation, which is originally carried out by AbsInt's decoder *exec2crl*, is skipped. The CRL2 representation created by llir2crl encompasses all required information. It is handed to the next application in the aiT toolchain that follows exec2crl in the ordinary workflow. The modified system is depicted in figure 5.1.

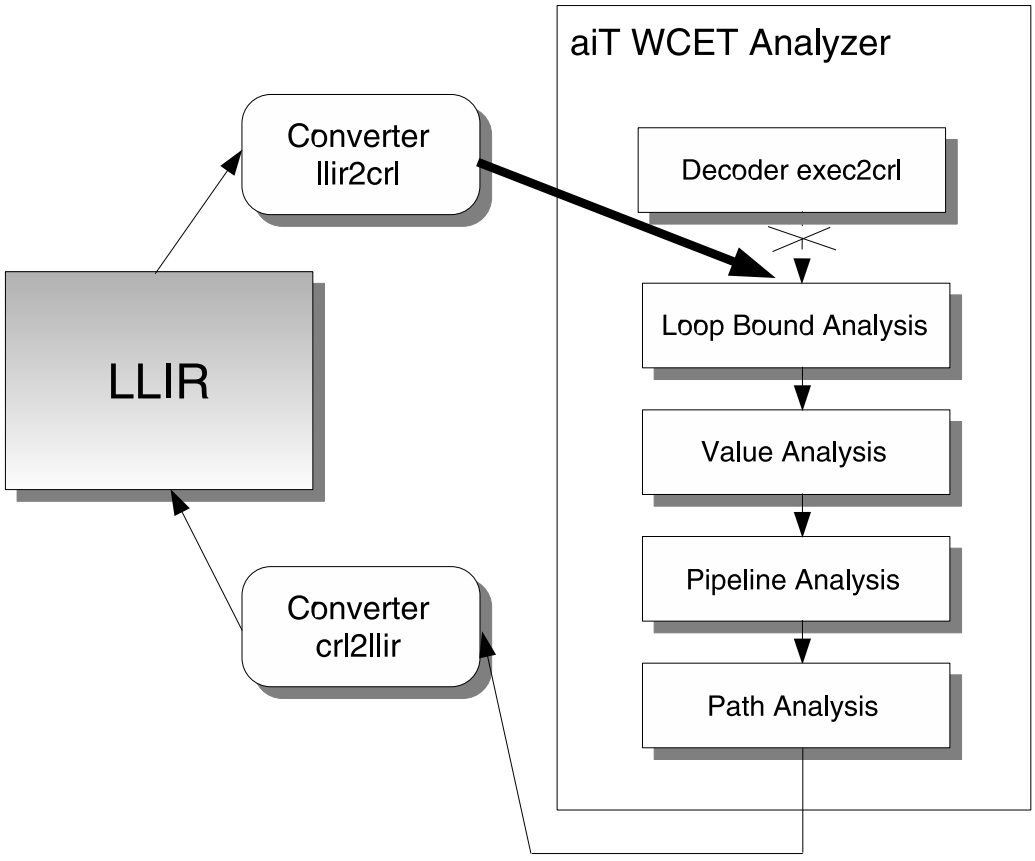


Figure 5.1: Modified workflow of the aiT WCET analyzer

5.2.1 Specifications and User Annotations

At the very beginning, the converter reads the configuration file called *wcetrc*. The purpose of this file is to provide external details which are beneficial or even inevitable for the CRL2 file that serves as input for aiT. Basically, this file corresponds to AbsInt's specification file, called *AIS*, that is provided with the binary executable to the aiT WCET analyzer. Numerous options, which can be defined in the AIS file, are not available for wcetrc since there is no use for them. These options mainly serve AbsInt's exec2crl to reconstruct the control flow from a binary program. Also, the syntax of the specifications and annotations used in both configuration files differs. The file used in this thesis operates on a simplified syntax, that still satisfies the requisites.

To enhance the analysis, the name of the compiler, which is used to translate the source program, can be defined. Different versions of compilers generate different patterns for specific source code elements, such as loops. Providing the compiler name, aiT examines its internal intermediate representation for known patterns which are supposed to be generated by a specific compiler. Recognized patterns protect aiT of a misleading interpretation and thus assure the correctness of the WCET results. Currently, the only supported value for the compiler name is `GCC Tricore C compiler`.

Furthermore, the clock rate of the underlying hardware can be defined. This results in WCET estimates which are not only given in cycles but also in real time units, e.g. seconds or milliseconds. This information is essential for the analysis of real-time applications which rely on the authentic timing estimates.

Another option, which is significant for the WCET analysis, is the specification of the memory areas of the underlying hardware. More accurately, memory area ranges that hold program code and data as well as their access times have to be declared. aiT relies on these specifications since it requires further details on memory addresses that are extracted by exec2crl. The user has to consult the manual of the underlying hardware to figure out the appropriate memory areas. The memory specifications are applied by the converter when new CRL2 components are generated. aiT does not demand the exact memory location of any element but necessitates the usage of valid addresses. For the very first element, which is usually a CRL2 routine, the lowest address range is determined and assigned as an attribute. Succeeding CRL2 elements are tagged adjacent memory addresses. Before an address is assigned, it is checked if not exceeding the given memory ranges.

The calculation of new memory addresses must take the instruction type into account. 16-bit instructions, such as the `CMOV` instruction (*Conditional Move*) [Inf05], occupy 2 bytes. In contrast, 32-bit instructions like the `ABS` instruction (*Absolute Value*) demand twice as much size, namely 4 bytes.

Loop bound annotations are another specification which is essential for the WCET analysis of programs that contain loops. Missing annotations cause a failure of the WCET analyzer, since many loop bounds can not be determined automatically. The loop anno-

tations are manifold. Global specifications, which concern all loops that are found in the source program, can be defined. The global bounds are defined by a minimum and a maximum representing the lower and the upper iteration bound, respectively. Thus, the loops are not considered individually but annotated with identical bounds throughout the entire program. This approach is adequate for some superficial tests in order to get a first notion of the timing behavior of the program. Moreover, it allows one to perform the WCET analysis at all since missing individual bounds lead to a program failure.

For more precise WCET estimates, the configuration can be extended by individual iteration bounds for each loop. aiT expects each loop to be specified by a routine name and the position of the loop within this routine. Each loop is enumerated, beginning with the decimal number one for the first loop. Thus, the user is supposed to consult the source code in order to determine the routine name and position of the loop he wants to specify. In addition, akin to global loops, the individual lower and upper loop bound specification has to be provided.

The manual annotations in terms of loop bound specifications represent a temporary solution. They turn out to be a tedious chore since great care must be taken to identify the appropriate loops from the source code. Furthermore, it is a difficult task to specify loop bounds at all since, in certain cases, the number of loop iterations can not be determined by solely reading the source code. The estimation of the worst-case execution time is still a relatively new field of research and most of the few existing WCET analyzers demand manual loop bound specifications. This annotation problem is well-known and has, for example, been subject of criticism by [Byh04].

A final solution gets rid of manual specifications. The intent of a sophisticated compiler framework is the support of automatic loop bound annotations. The internal mechanisms of the framework recognize loops, analyze them and finally provide iteration bounds which are automatically used in the low-level intermediate representation. A loop analyzer is a complex application and thus beyond the scope of this thesis. However, it is intended to realize this feature in the future compiler framework where the software modules developed in this thesis will be integrated in.

Besides the loop bounds, two other options, concerning the applied contexts, influence the analysis process. The parameter `CALL_STRING` restricts the length of the call string, e.g. it defines whether all possible context branches or just a limited number should be taken into account. Permissable values are decimals or the specifier *inf* which is short for infinite. For complex source programs, a tradeoff between the precision of the estimates and the analysis time is required. Longer call strings produce more precise results but also increase the runtime of the analysis. Hence, best results are obtained with the parameter value *inf* but they might entail an unacceptably long analysis time. A related configuration parameter is `UNROLL`. It defines the number of loop contexts to be distinguished. Especially the first loop iterations differ in terms of their cache behavior. The very first loop iteration yields a cache miss since no loop data is available in the cache memory yet. This data has to be fetched from the main memory. Due to speculative prefetching,

it might happen that the cache content is still modified during the second loop iteration, and another load of data into the cache is necessary. For further loop iterations, the cache situation gets stabilized and does not differ remarkably. Thus, it makes sense to distinguish at least among the first loop iterations. To do so, the parameter `UNROLL` is assigned a decimal, representing the number of individual loop contexts. For example, `UNROLL = 3` means that the first and second loop iteration are examined separately and the subsequent iterations are combined into the third loop context.

The options mentioned above, which concern loops, are taken into account when the LLIR is converter into the CRL2 representation. They are temporarily added to the CRL2 routines that contain loops. As will be described , the initial version of the CRL2 representation is modified by the loop transformation which relies on these annotations.

5.2.2 Transformation of the CFG Structure

After processing the configuration file, the transformation of the LLIR structure to CRL2 is performed. As stated in chapter 3 and chapter 4, the structure of both the LLIR and the CRL2 is highly analog in terms of the involved key components. Both intermediate representations employ functions (called routines by CRL2), basic blocks, instructions, operations and their operands, i.e. registers or constants. Thus, the process of the conversion between the LLIR and the CRL2 is mainly based on the traversal of all components in the form of nested loops.

First of all, the outermost loop iterates through all LLIR functions. They are derived from functions that are used in the source program and bear the same name. Concerning the fact that each LLIR function corresponds to a CRL2 routine, the converter creates a new routine whenever a new LLIR function is analyzed. This new CRL2 routine is assigned the same name as the LLIR function to keep track of both elements throughout the analyses and conversion.

The next element in the hierarchy of a program structure are basic blocks. They represent a maximal group of statements so that one statement in the group is executed if every statement is executed [KA02]. Each new consideration of an LLIR basic block leads to the creation of a new CRL2 basic block. Like routines, each CRL2 block is denoted by a unique label derived from the LLIR basic block. According to the definition mentioned above, function calls finalize a basic block if the function may not return. Thus, two contrasting block structures may occur. In the high-level language C, it can not be predicted at compile time, whether a function will return or not. The only way is to look it up in the source code. Hence, conservative compilers assume that a `call` instruction will end the basic block. In order to avoid a break of the control flow, the successive blocks are combined by a CFG edge (see figure 5.2). Unlike C, the language C++ offers the opportunity to guarantee that a function will return to its caller by marking it with `throw()`. In that case, the low-level intermediate representation is not forced to split the basic block

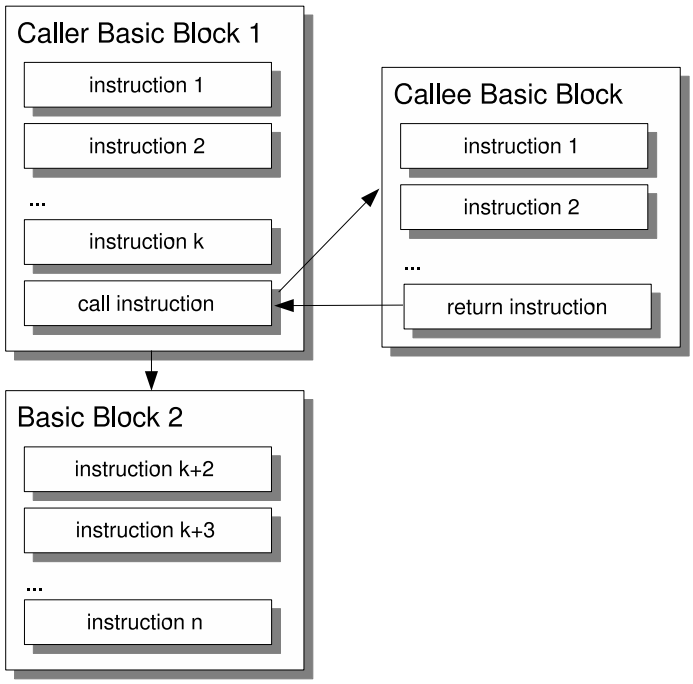


Figure 5.2: Call instruction ends basic block

after the `call` instruction, but may keep this instruction amid the other instruction within the block. Figure 5.3 gives a pictorial view of the caller-callee relationship. Since aiT is expecting input files written in C, the former block structure has to be applied.

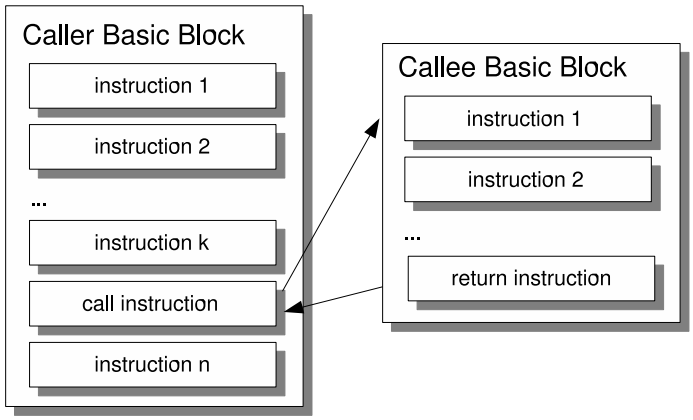


Figure 5.3: Call instruction amid basic block

Next, the instructions are processed. Another loop walks through all LLIR instructions of a basic block and creates a CRL2 instruction. Instructions do not have any unique identification like functions or basic blocks. However, they need to be relocated in the further steps. For instance, the value analysis performed by aiT assigns potential value ranges to each CRL2 instruction. It is highly desirable to import these results into the LLIR. To keep

track of the original LLIR instructions and the belonging CRL2 instructions, an adoption of a unique identification is essential. To solve this problem, LLIR pragmas are employed. They serve the purpose of providing additional data that can be tagged to almost any LLIR component. The information held by the pragmas is either an arbitrary string, which can be generated by the user, or a predefined information that was defined in the initialization phase of the LLIR. Every LLIR instruction is labeled with a unique string, consisting of a prefix and an individual decimal number. The same label is added as attribute to the CRL2 instructions. Thus, at any point of the aiT analysis, the corresponding LLIR instruction can be easily determined. This track-keeping is notably relevant for the conversion of analysis results from the CRL2 to the LLIR which is carried out by the converter *crl2llir*.

At the bottom of the hierarchy, LLIR operations and parameters are analyzed. This repetitive process is performed by the two innermost nested loops. Depending on the target hardware, the examined instruction might encompass one or several operations that are executed in parallel. Each new LLIR operation causes the creation of its counterpart, the CRL2 operation. Operations are not denoted by a unique identification. Unlike instructions, aiT does not produce any analysis results which would concern individual operations. So, at any time, there is no need to revert to a specific LLIR operation; therefore their unambiguous specification is deemed redundant.

Each operation is comprised of resources that are involved during the execution. In contrast to the CRL2, the LLIR handles these resources in the form of a separate class. The operation class in turn distinguishes among different types of parameters. Plain types are covered by constants and labels, which merely hold their actual value, mainly an integer number or a character string, respectively. A more complex parameter is a register which is modeled by its own class. It does not suffice to know the name of the register. Compiler backend analyses and optimizations rely on different preferences of used registers. For example, the *Lifetime Analysis* and the *Def/use Chain Analysis* (see section 4.2) investigates the usage type of registers at a specific point, e.g. the registers are checked if they are employed for read accesses, write accesses or even both. These characteristics as well as some others are modeled as class attributes.

As mentioned above, the CRL2 format does not provide any classes for parameters. Constants are handled by internal data types that basically express numerical values. Likewise, labels are represented by internal numerical data types. Due to the fact that the CRL2 library assumes a binary program as input, the labels are not treated as strings but as addresses which got resolved by the linker. And finally, registers are modeled by internal string types that hold their name. At first view, the representation of operation parameters by simple data types might seem to impose some restrictions. However, CRL2 is cooperating with AbsInt's NET library that provides a complex data base for each instruction of a specific processor. Based on this data, it is sufficient to supply the NET library with the register names in order to construct fully detailed CRL2 instructions. An in-depth description of the powerful NET library and its functions will follow in the next subsection.

With respect to the classification of operation resources, both intermediate represen-

tations act in the same manner. The LLIR as well as the CRL2 distinguish between resources which are exclusively read or written. The LLIR uses the notation *USE* and *DEF* for read-only and writable resources, respectively. AbsInt's CRL2 format designates resources which are read by *src* and those which are written by *dst*. Due to the nature of constants and labels, these operation parameters are solely marked as readable preventing write accesses.

In addition to the registers, which occur in an assembly file, most low-level intermediate representations operate on internal auxiliary registers that reflect the actual structure of an operation more precisely. Moreover, these registers are used for the data flow analysis in order to model dependencies which do not emerge with ordinary assembler registers. Thus, they are essential for numerous optimization approaches which rely on the results from the DFA. These internal registers are processor-dependent and must be predefined for each target architecture. For instance, for the Infineon TriCore TC1910 DSP, the LLIR allows the creation of the internal registers *CSA* and *CSFR*. The former denotes the *Context Save Area*. It designates the memory area that is used by the processor to store a current context. The memory area may consist of different processor registers which are statically defined for each individual architecture. The CSA is employed for context and task switching. For example, the TriCore instruction set architecture (ISA) involves the Context Save Area in its `CALL` instruction to store the current register values and jump to another subroutine. After the return to the caller routine, the saved values are loaded and the previous state of the program is restored. The internal register CSFR addresses to the *Core Special Function Register* which is a particular register that can only be accessed by some few specific instructions, e.g. the instruction `MFCR` (Move From Core Register) involves CSFR at the startup of the executable code.

These internal registers are omitted when it comes to the conversion of the LLIR into the CRL2 format. The reason is that the NET library is aware of any resources that are used with a particular operation. Since the internal registers are constant for a specific instruction, they are not required to be known when a new CRL2 operation is created. In contrast, the regular data and address registers, which are found in an assembly code, differ since they represent an individual program. Depending on the prior assembly code, the register allocator replaces virtual registers by physical registers that are currently not used. Thus, to reflect the actual code, the involved register names have to be supplied.

5.2.3 The TF14NET Library

The aiT WCET analyzer performs its analyses based on AbsInt's low-level intermediate representation. This representation is called CRL2 and represents a versatile processor-independent format (see chapter 2). To enable analyses for a particular processor, CRL2 requires further hardware information. This data is given in a so-called *NET* file that represents a unique specification for a certain processor. Due to the complexity of present-day processors, the NET file possesses an elaborate structure. To simplify the retrieval of data,

the *tf14net* library is applied. It can be interlinked with any processor specification file and provides functions to read the NET file as well as to evaluate the gained information. A prominent example is the collaboration with the decoder exec2crl that makes extensive use of the library to construct a CRL2 control-flow graph from a binary executable program.

In this thesis, the tf14net library is used with the TriCore NET file. It cooperates with the CRL2 library in order to generate appropriate CRL2 instructions. In addition to the CRL2 library, both the tf14net library and the TriCore NET file have been provided by courtesy of AbsInt.

The NET file holds numerous information on the processor architecture. In the very beginning, possible register types are declared. Besides their classification in data, address and extended registers, each register type is specified by its width and composition. In addition to some other auxiliary declarations, each operation that is provided by an individual processor is described in detail. The description is divided into two sections. The first one encompasses attributes that are valid for the entire operation, e.g. the opcode, the mnemonic, the addressing mode and the number of execution cycles. The second one consists of specifications that concern single operands.

The attribute opcode, which is called *op_id*, serves the purpose of identifying an operation. It represents a unique binary number. Within this binary representation, which can at most have 64 bits, the corresponding machine code bits are encoded. To put it in other words, each operation is assigned a predefined unique number that is stored in the attribute op_id. Depending on the target architecture, the structure of this binary number may differ. Some processors read the machine code byte-wise, others word-wise. Also, the byte order scheme, namely little-endian and big-endian, affects the composition of this unique identification number.

The usual workflow of retrieving the appropriate operation from a binary executable program begins with the decoder exec2crl. Depending on the architecture, each operation obeys a specific format called the *operation format*. The formats consist of groups of bytes that represent either the operation ID or the used operands. Additionally, the global attribute *mc_bitmask*, which is defined in the NET file for each operation, is required. Again, it is a binary number which is used as a mask to determine the significant bits of the attribute op_id. More precisely, each position of bits of mc_bitmask that are set to `1` is also examined in the attribute op_id. If both attributes hold the value `1` in same positions, a match is achieved which indicates that the appropriate operation was found. In the case of a deviation, the considered operation does not match and the recognition algorithm continues with the next one. This repetitive comparison is applied to the entire byte stream of the binary program [The01].

If a valid operation is found, the decoder is still compelled to supply the involved resources. Having all mandatory data available, the tf14net library starts to construct the actual CRL2 operation. This process is highly complex and emphasizes the full strength of the library. Each complex operation in the NET file is assigned a set of extensions. Ex-

tensions are a hierarchical description of operands. Each operation may be comprised of several operands which are accessed in a well-defined order. Furthermore, each operand in turn may even have a complex structure which needs a hierarchical representation. The complex structure as well as the access order are modeled by extensions. Due to the diversity of possible operands, operations contain groups of extensions. The same bit-matching algorithm, which carried out the identification of operations by their op_id, is responsible for the selection of the proper extension. Together with the register names, the tf14net library is able to create a full CRL2 operation. Exactly this construction chore is done by AbsInt's decoder automatically. The byte stream extracted from the binary executable is analyzed and passed to the tf14net library. The result is a sequence of CRL2 operations which contribute to the reconstruction of the control-flow graph.

5.2.4 Instruction Recognition

The most appropriate way to create CRL2 instructions is to employ the functions which are provided with the tf14net library. Like exec2crl, the developed converter llir2crl incorporates library functions to generate CRL2 instructions which are added to the corresponding CRL2 basic blocks. The functions require two sets of data, namely the op_id and the involved resources.

Within the TriCore NET file, each supported operation is designated by a unique op_id. The classification of the operations rely on the mnemonic, the alignment of the operands (called *opcode format*) and the addressing mode. In contrast, the LLIR merely accommodates mnemonics and the addressing mode but does not hold any direct information on the opcode format. Due to this missing information, a direct mapping of NET operations and LLIR operations in not possible. For example, the class of `AND` operations provided by the TriCore ISA encompasses 4 different unique operations. They are characterized by their mnemonic and their opcode format. The addressing mode, as for most operations, is not given, since the operations are not involved in the calculation of effective memory addresses. In the current state of the LLIR, a direct mapping to the NET operation is not unambiguous. As can be seen in figure 5.4, an LLIR operation with the mnemonic `AND` could correspond to four different NET operations.

A solution to this problem is the determination of opcode formats based upon LLIR data. For this purpose, the types and the alignment of operation operands must be analyzed. The detailed proceeding will be described later.

Operations, which operate on memory addresses, are characterized by their addressing mode. For example, the TriCore ISA supports 12 individual `LD.W` (load word) operations. Five of them can be identified by the opcode format, the remaining ones are assigned a unique addressing mode. Thus, besides the opcode format, the addressing mode, as further search criteria, has to be taken into account.

AbsInt's decoder exploits the full range of data that is given in a binary executable.

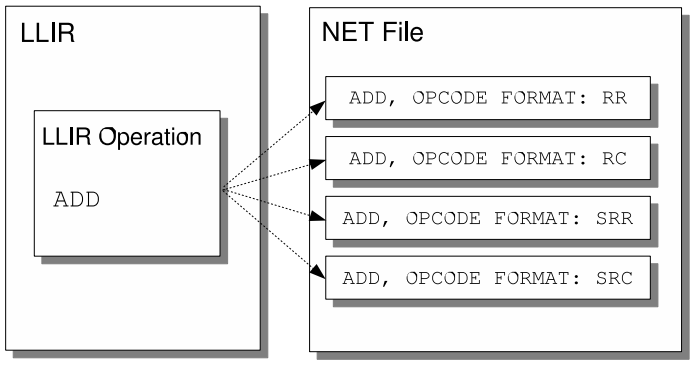


Figure 5.4: Ambiguous assignment of LLIR and NET operations

Each byte of the program can be analyzed; this is extremely beneficial when it comes to the recognition of the op_id. The read bits of the byte stream are examined by a bit-matching algorithm in order to obtain the right operation. Thus, it is not required to analyze each operation parameter to find the corresponding operation format. Since the LLIR does not hold any information, which were gained from a program that was run against an assembler and linker, the full information range from the byte stream is missing. Hence, the given LLIR information serves as the resource to find the appropriate operation identification.

For the purpose of finding the proper op_id, an algorithm has been developed. It requires the LLIR operation, which is supposed to be analyzed, as well as the NET file. In the first phase, the significant data from the NET file is transformed into an internal data structure in order to omit any data, which is not relevant, and thus to simplify the evaluation. The attributes from the NET file, which are involved in the algorithm, are: the mnemonic, the operation format, the addressing mode and finally the sought op_id. Mnemonics represent the human-readable form of on opcode, e.g. `ADD` or `ST.W` (store word). Although they substantially restrict the search for the actual operation, they still do not yield unambiguous operations. This ambiguity results from the fact that there might be several operations which are assigned the same mnemonic.

To get a more subtle specification, further operation characteristics have to be considered. The next attribute, which again decreases the number of potential candidates for the sought-after operation, is the *opcode format*. The TriCore DSP distinguishes between two classes of formats. As the attribute names suggest, the 16-bit and the 32-bit opcode formats are incorporated in 16-bit operations and 32-bit operations, respectively. Each class contains numerous formats which rely on the used parameters. For example, the TriCore `ADD` instruction encompasses 8 different opcode formats which indicate whether the instruction has a width of 16 or 32 bit. Moreover, the formats differ in terms of the involved parameters; they state if constants, regular data registers or even the implicit ata registers `D15` are used. The following listing illustrates some typical TriCore ADD instructions which are assigned different opcode formats:

```
# 16-bit opcode formats
```

```
ADD D3, D2
ADD D3, #126
ADD D15, D2, D3

# 32-bit opcode formats
ADD D3, D1, D2
ADD D3, D1, #126
```

Here, `D1`, `D2` and `D3` are regular data registers, whereas `D15` represents the implicit data register [Inf05].

The mnemonic and the opcode format are almost sufficient for an unambiguous identification of an operation. However, there are still some instructions which have an identical mnemonic and operation format parameters. The only distinction is their varying addressing mode. Most machine languages refer to operands that are stored in memory. The addressing mode specifies the calculation of an effective memory address of a certain operation operand, using values held in registers and constants. Mainly load and store instructions make extensive use of the various addressing modes to operate on the memory. The used target architecture in this thesis provides eight different modes such as the *Pre-* and *Post-increment Addressing Mode* or the *Short* and *Long Offset Addressing Mode*.

To sum it up, the first phase of the opcode recognition algorithm reads the mnemonics, the opcode formats, the addressing modes and the operation IDs from the NET file. These attributes are stored in an internal data structure which is implemented in the form of nested maps. The outer loop stores all mnemonics as map keys. The corresponding values are again maps which accommodate the opcode format as a map key and refer to the core map as a value. The core map as the innermost component stores addressing modes as map keys and the sought operation IDs as map values. The data structures employed by the algorithm are depicted in figure 5.5.

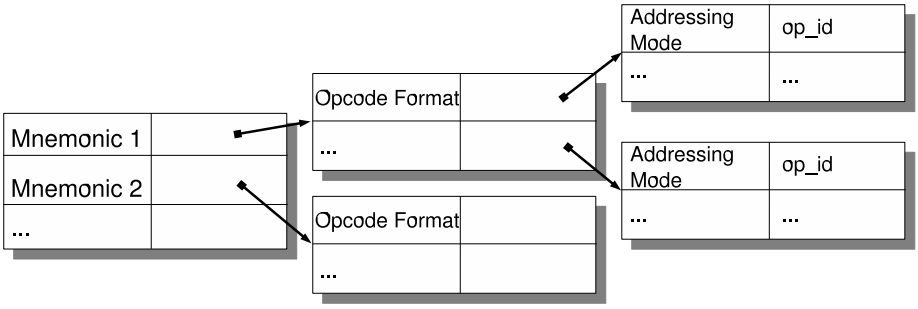


Figure 5.5: Overview of the internal data structure holding the operation ID

However, there are still some exotic operations which can not be unambiguously identified. Notably multiplication operations correspond in their mnemonic and their operation format, but are assigned no addressing mode. They differ exclusively in a specific identifier which indicates one of the possible

multiplication modes. For example, the TriCore processor supports four cases: *Upper * Upper, Upper * Lower, Lower * Upper* and *Lower * Lower* (see [Inf05]):

```
MADDR.H D[c], D[d], D[a], D[b] UU, n #opcode format (RRR1)
MADDR.H D[c], D[d], D[a], D[b] UL, n #opcode format (RRR1)
MADDR.H D[c], D[d], D[a], D[b] LU, n #opcode format (RRR1)
MADDR.H D[c], D[d], D[a], D[b] LL, n #opcode format (RRR1)
```

The syntax tokens `D[a] - D[d]` designate data registers while `n` symbolizes a *multiplication result shift value*. As can be seen, all four operations bear the same mnemonic `MADDR.H` (Packed Multiply-Add Q Format with Rounding) and are denoted by the same opcode format (RRR1). The only difference are the identifiers `UU`, `UL`, `LU` and `LL` which occur as operation operators in the assembly code. Due to the fact that the total number of these differently structured operations is relatively small, their recognition is static, i.e. the algorithm data structures are not consulted but the appropriate op_ids are hard-coded.

In the second phase of the algorithm, the structure of each LLIR operation is analyzed. Firstly, the mnemonic is determined and employed as search criteria for the outer map of the internal data structure in order to retrieve the inner map that holds the operation formats. Secondly, a set of rules is consulted to find the appropriate operation format. The rules refer to the operation operands which, depending on their type and alignment, yield a specific format. During the development of the algorithm, one of the main efforts was the creation of the rule base. A thorough examination of each opcode format provided by the TriCore DSP was performed. Firstly, different operand types had to be taken into account. The main distinction relates to the basic operand types that are supported by the LLIR, mainly registers, constants or labels. However, this superficial differentiation is not sufficient to identify a unique format. For this reason, a more subtle analysis has to be utilized. Further distinctions refer to the width of constants and the characteristics of registers. For some formats, it is crucial to know whether a constant has a certain width, e.g. 9 or 16 bits. Another decisive factor is the register type; among the distinction of data and address registers, some formats are up to the precise register description, e.g. the TriCore implicit data register `D15`.

5.2.5 Assembler Directives

Together with the hard-coded part of some exotic multiplication operations, the algorithm has the ability to identify nearly all operations unambiguously. However, the TriCore instruction set still contains few operations which can not be determined explicitly with the means provided by the LLIR. For instance, there are two versions of the `MOV` instruction:

```
# 32-bit instruction
MOV D[c], D[b]

# 16-bit instruction
```

```
MOV D[a], D[b]
```

`D[a]`, `D[b]` and `D[c]` denote arbitrary data registers. Within the LLIR, an operation of this type merely holds two registers as operands. The registers are assigned a regular name that is a concatenated string of the character *D* (designating a data register) and a decimal number, ranging from 0 to 14, as suffix. The cause of missing detailed information emanates from the nature of the LLIR. Due to the fact that the LLIR is supplied with an assembly file as input, there are no precise information about the involved operations. The crux of the problem is that the assembly code does not expose any details on the operation width. Thus, it can not be inferred if a 16-bit or 32-bit instruction is utilized.

This ambiguity prevents a proper generation of CRL2 operations. The NET file contains unique op_ids for each width-dependent operation. However, when it is unknown if the assembly code represents a 16-bit or 32-bit operation, the op_id can not be exactly defined. Incorrectly chosen operation IDs would yield CRL2 operations which do not reflect the actual LLIR code (and thus the program to be analyzed). To prevent deviations between both the LLIR and the CRL2 representation, consistent assumptions must be made. The instruction set of the TriCore processor provides for the ambiguous 16-bit instructions equivalent 32-bit instructions but not vice versa [EF94]. Thus, it will be assumed that each ambiguous operation is handled in its a 32-bit version. Accordingly, care must be taken that the generated CRL2 operations are employed in their 32-bit version. Consequently, the assembler has to be forced to translate these assembly operations in appropriate 32-bit machine code operations.

In this manner, a synchronization between the LLIR and CRL2 is achieved. It is guaranteed that the CRL2 representation, which is used as input to AbsInt's aiT WCET analyzer, operates on the same operations which are dumped out by the LLIR. This synchronization is essential for proper timing results. To make things clearer, assume that the treatment of ambiguous operations was omitted. It could happen that the converted llir2crl creates a CRL2 operation that reflects the 16-bit version. On the other hand, the LLIR produces assembly code and this code is handed to the assembler. Without any directives, the assembler assumes the operation to be 32-bit and generates the corresponding machine code. The result is a deviation between the actual binary program and the program analyzed by aiT. Since the analyzer is operating on different operations, the calculated worst-case execution estimates do not reflect the authentic timing behavior of the compiled program.

Due to the same reason, the overall burden of the tedious operation identification is mandatory. At first view, it might seem that it would suffice to take the mnemonics exclusively into account. Based on this information, one could easily select an arbitrary op_id that is assigned to this mnemonic. Certainly, aiT would produce WCET estimates which approximately reflect the actual timing estimates of the binary program. As mentioned above, a deviation would arise from varying operations used for the program and for the output to the analyzer. However, when precision is an issue, both the analyzer input and the machine code of the compiled program must be identical. To put it in other words, both

constructs must utilize exactly the same operations.

To solve the dilemma of unambiguous operations, *assembler directives* (often called pseudo-operations) are employed. In addition to regular statements for machine instructions, present-day assemblers provide directives to steer the assembly process. To differentiate between them from the regular assembly instructions, the directive names are prefixed by a period. A large number of the assembler directives is processor-independent. For instance, the directive `.global` makes a symbol visible to the linker and may be notably applied to handle global variables that are given in a source code. Global variables, like in the high-level language C, are defined outside any function and may be accessed from other classes. To accomplish a global scope of a global variable, other partial programs, which are involved in the linking process, have to be aware of that variable. This is exactly done by the directive `.global`.

Other assembler directives are tailored to a specific target hardware and exploit their characteristics. The TriCore assembler comes with some specific directives to steer the translation process. `.code16` and `.code32` coerce the assembler into emitting a 16-bit or 32-bit opcode for the subsequent operation, respectively [EF94]. These directives are utilized in this thesis work to map out a specific operation in order to avoid ambiguity.

Whenever an operation representation is detected in the LLIR, which can not be exactly determined, the LLIR is extended by a new pragma. Usually, LLIR pragmas are employed to provide additional information on the key components, e.g. they are used as comments that are passed to the assembly code. For this thesis, the pragmas have been slightly modified to serve the purpose of assembler directives. They are added to the LLIR and occur in the assembly code that is generated by the LLIR as output. This code can be supplied to the assembler which generates machine code that contains the predefined operations.

Of course, the algorithm, which is responsible for the generation of the proper op_ids, takes the aforementioned special cases into account. Depending on the existence of appropriate assembler directives, the proper op_id is generated and passed to the tf14net library which finally creates the sought CRL2 operation.

5.3 Required Extensions to the LLIR

This section discusses the extensions to the Low-Level Intermediate Representation (LLIR) that are required to manage additional information on its elements. In the current state, the LLIR serves the purpose of a pure compiler backend for regular systems that do not rely on a precise execution time. The LLIR provides all essential data structures to construct a control-flow graph of a given program and to perform data flow analyses. Together with the available optimization techniques, the LLIR allows one to produce effective assembly code which provides an ample basis for further code transformations.

Unlike regular systems, real-time systems are time-critical platforms which rely on the

observance of execution time bounds. In order to generate code for this type of systems, the compiler backend requires details on the timing behavior of a program. This inevitable information must be available for basic blocks which represent an atomic unit in regard to the low-level consideration of a program.

It is desired to create a compiler that offers the opportunity to produce code for real-time applications. To accomplish this goal, the current compiler framework must be extended. More precisely, the extensions concern the low-level representation of the program to be analyzed which is given by the LLIR. The extensions consist of new data structures that accommodate timing information but also of appropriate mechanisms to read and write this data.

The primary goal of this thesis is to design and realize concepts for worst-case execution time compiler optimizations. That is, the LLIR must provide WCET details on its structural components. Basically, timing information is tagged to basic blocks since they are regarded as an atomic unit which is executed entirely or not at all. However, for the sake of simplicity, it is beneficial to provide WCET information also for LLIR components which are on a higher hierarchical level. For example, LLIR functions, which are comprised of at least one basic block, should expose the accumulated WCET of all their basic blocks. Certainly, this goal can be achieved by examining all basic blocks separately and adding their local worst-case execution times. This approach is very cumbersome, though, and is therefore replaced by transparent functions which return the accumulated timing estimates for a specific LLIR function. The performed calculations which are behind this `get` function are hidden from the user.

Based on these extensions, the goal pursued in the present thesis can be accomplished. The supplementary WCET estimates serve as basis for a compiler that optimizes code in terms of the program execution time. By integrating the LLIR extensions and the various software add-on modules, which have been developed within this thesis, one can develop novel analysis and optimizing algorithms. Due to the simple interface, which will be described in more details in section 5.4, the user can fully focus his work on the development of the algorithms and is untroubled by a complicated retrieval of required data.

While designing the concepts for the extensions to the compiler framework, attention was paid to both reusability and generics. Although the main goal of this thesis was to extend the LLIR in terms of the worst-case execution time, the involved concepts allow an easy upgrade to the backend software for other applications. This is made possible by the use of an unchanging interface which is responsible for the connection between the LLIR elements and the supplementary data.

The general idea is shown in figure 5.6. To achieve a separation between the LLIR and the additional information, the concept of objectives is adopted. A thorough introduction to objectives will be given in section 5.4. Objectives are individual units that store various information, e.g. the worst-case execution time, the best-case execution time (BCET) or the energy consumption. The structure of the objectives is not predefined and can be

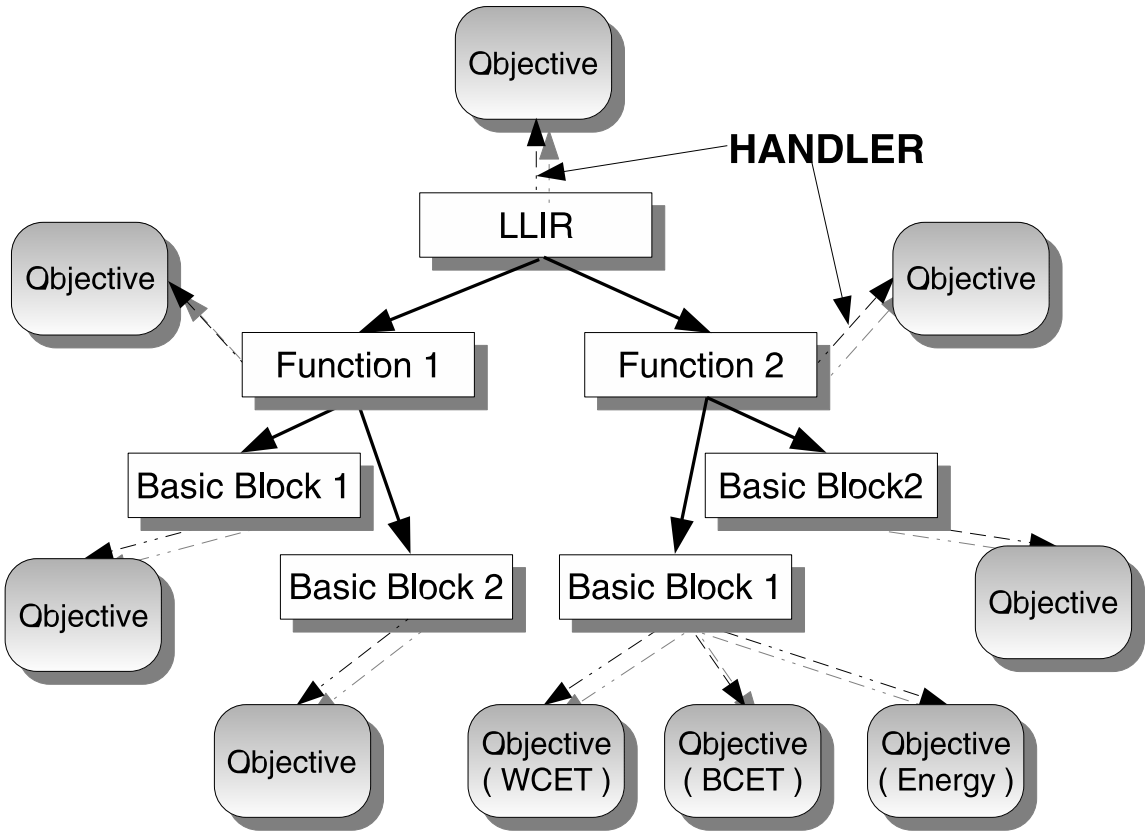


Figure 5.6: Interaction between LLIR and objectives

modeled arbitrarily depending on the given requirements. The user may create new types of objectives which are tailored to a specific application. These objectives can have user-defined functions and attributes. Each LLIR component is equipped with a *handler* which connects the former with an arbitrary number of objectives. This means that the supplementary information is not directly stored in the LLIR but handled by a connection to the information carriers.

Assume that the LLIR is supposed to be involved in a foreign application field which demands new details to the LLIR components. Instead of modifying the LLIR code, new tailored objectives are realized which can be accessed by the handler. Thus, the LLIR represents a central point for the retrieval of new information but remains unchanged by itself. The entire compiler framework can be easily adapted to new applications by merely providing the appropriate objectives. This circumstance serves the reusability and is highly beneficial for the entire development process.

A huge advantage is given by the fact that the user is free to add or modify objectives that are suited to his needs. The design process succeeds independently of the LLIR structure. This circumstance fosters the simple and wide use of the objective framework.

Another aspect concerns the usage of the compiler framework by third parties who are

not permitted to view the source code of the LLIR. If the availability to the LLIR code was a prerequisite to produce new objectives, the entire framework would not be adequate for release purposes. The circle of potential developers would be substantially restricted to those who have full access to the source codes. Of course, this is rarely desired since the LLIR should remain *intellectual property*.

In contrast to the limited concept described above, the developed mechanism of this thesis allows an independent connection of customized objectives to the LLIR. The latter may be provided in a compiled version protecting the program code. Third parties are free to add new objectives that suite their needs without taking the LLIR directly in consideration. Hence, this design concept makes the LLIR a multifunctional backend that offers a wide range of application areas.

There is a great number of further potential applications where objectives, which serve as auxiliary information carriers, might be employed. In addition to the worst-case execution time, a related problem is given by the *best-case execution time* (BCET). Notably control systems may make extensive use of it. Often, lower time bounds have to be obeyed when the control center communicates with other units, i.e. the transmitted signals must not be received before a defined point of time [Erm03]. Due to the intensive correlation between the BCET and WCET, the development of appropriate objectives does not imply a high design and programming effort. The present WCET objectives can be basically reused, if not even adopted almost unmodified. Like the WCET estimation, sophisticated BCET analyzers perform their calculations on data expressed by context-dependent execution counts and local execution times. To generate precise results, the consideration of individual contexts for loops is essential. Since all data structures that store these analysis results already exist for the WCET objectives, they can be reused unchanged for the BCET issue. The remaining task would be the realization of an application that reads the results of the BCET analyzer and transforms them into the objectives.

Another field of application for objectives can be the optimization in terms of energy consumption. Especially in the domain of portable embedded systems, like cellular phones or pagers, batteries become a bottleneck. Their low performance prevents the reduction of system cost and weight. To overcome this dilemma, low power design techniques have to be applied [FV04]. Objectives, which hold appropriate energy information, would aid the process of developing novel optimization algorithms. To accomplish this goal from a simplified view, two problems have to be regarded.

Firstly, an energy model of the target hardware is required. For the sake of completeness, it should be mentioned that this information is yet not available for every processor. Some hardware models are deemed confidential and not published by their vendors. Among other details, the models expose the energy consumption of each instruction. This kind of data is ideally suited to be stored in an objective. Individual energy consumptions are wrapped in objectives and interlinked via handlers to the corresponding LLIR instructions. At this point, the strength of the developed handler-objective framework is once again accentuated. The required data from the energy model which requires to be

provided to the compiler backend does not entail any tedious modifications to the LLIR. The objectives are developed independently and can be easily combined with instruction objects. Also, the retrieval of the energy consumption details is simple due to the generic functions.

The second problem that is inevitable for the energy-aware optimizing algorithms is the specific order in which the instructions are executed. It turns out that different execution orders significantly affect the power consumption. In order to pay attention to this circumstance, individual objectives have to be set in mutual relation to interchange data. Thus, the execution order can be modeled. In summary, it can be ascertained that the objective framework is convenient as a basis for the development of energy consumption optimization algorithms.

To refer back to the concept discussion, the separation between the objectives and the LLIR eases the software maintenance. Both parts of the framework are developed autonomously. New objectives are added to the LLIR which can even be available in a compiled version. Similarly, the LLIR code can be maintained separately without any knowledge of the existing objectives. This permits the incorporation of independent LLIR versions into various projects. The developers expand their objective framework and connect it to their version of the LLIR. Another developer team may work on the LLIR code and provide new updates to the objective developers. Thus, a project structure with standalone teams is achieved. None of the developer teams is constrained by others. In the case of one single version of the LLIR, which served as basis for the integration of new objectives, a persistent synchronization would be mandatory. Each modification to the LLIR performed by one team had to be taken into consideration by the remaining developers. To keep the LLIR code consistent, each new modification would entail tedious consultations.

The handler is not only utilized to read objectives. The converter *crl2llir* creates new objectives that accommodate the results produced by the aiT WCET analyzer. Due to varying requirements, each type of an LLIR element expects a specific objective. After the creation of the objectives, the corresponding handlers are determined and the new objectives are appropriately interlinked.

In addition to the reusability, great attention was paid on generic concepts. The propelling idea was to provide a universal interface between the LLIR and the objective framework which was valid for any type of objectives. As already stated in the discussion on reusability, a strict separation between the LLIR and the objectives was pursued in this thesis. Due to this fact, the mechanisms for the retrieval of information that are stored in the individual objectives must remain constant. It is desired to have one identical `get` function that is contained in each LLIR object. This function should have a unique name and an identical return type.

As a result, the consistent functions simplify the access to the objectives. Each LLIR object contains a function called `getHandler`. The function is derived from a base class and is thus identical for all objects. Its purpose is to retrieve the corresponding handler

which grants access to the objectives that are available. In this manner, a connection between the LLIR components and their information carriers is established and the additional data might be read.

The main advantage of one single function is that the user does not need to care about individual function calls. Regardless of the current LLIR element, there is invariably one function which retrieves the corresponding handler that in turn reads and writes the objectives. If the handler was omitted and each objective was directly attached to the LLIR elements, then a confusing class structure would emerge. For each new objective type, individual interface functions were required. Moreover, all objectives which were ever added to the LLIR, had to be available in order to guarantee a proper operation mode of their LLIR functions. To jettison obsolete objectives, a tedious modification of the LLIR code would be required. In contrast, the elaborate handler that was developed in this thesis provides a function to check if a sought objective is available for a specific LLIR element. Thus, the user is given an overview of all present objectives. However, if an objectives no longer exists, there are no individual LLIR functions which refer to the missing objectives. Hence, the removal of unused objectives does not induce any modifications to the LLIR code.

To achieve a standard access to the objective handler, each LLIR class must hold the same functions. The most efficient way to accomplish this goal is to add the handler function exclusively to one class which serves as basis for all other classes. The high-level language C++, which is used to implement the LLIR, provides a form of software reusability called *inheritance*. The programmer creates classes that absorb an existing class' data and behavior and enhance them with new capabilities. This existing class is referred to as the *base class*. The inheritance mechanism is exploited for the realization of the handler function. Each LLIR class derives its members from the class LLIR_TaggedElement. In the past, this base class was employed to summarize multiple LLIR pragmas. The latter are stored in a list of the base class and passed to the derived classes. As a result, the pragma list requires to be created once and is automatically available in all newly created LLIR classes. In the same fashion, the function `getHandler` is appended to the class LLIR_TaggedElement. Hence, all LLIR components, except the register class which can not be assigned pragmas, obtain an interface to the objectives.

It has been shown that the elaborate handler mechanism developed in this thesis benefits both the reusability of the LLIR and the simplified access to the objectives. On the one hand, the LLIR might be involved in independent projects which design and realize user-defined objectives. On the other hand, the use of inheritance grants a standardized interface to the objectives; thus read and write accesses can be easily performed.

5.4 Objectives and Handlers

The developed mechanism of extending LLIR elements by further data is based upon handlers and individual objectives. The latter are information carriers which are interlinked with LLIR elements via handlers. They can be customized for various applications since their structure is not subject to any restrictions.

A major task in the initial phase of this thesis is to design an appropriate concept for supplementary information carriers. The primary goal concerns backend extensions in terms of the worst-case execution time. However, the developed concepts should not merely serve this single purpose but are supposed to provide a versatile utilization. The goal of the extensions to the current compiler framework is a novel interface that can be reused for further applications. More accurately, the implemented mechanisms should remain unmodified when new types of objectives are added. Besides their independent development, a versatile approach for interlinking with LLIR elements should be supported. To realize these requirements, various concepts have been taken into account. In the following, the evolution process is briefly outlined.

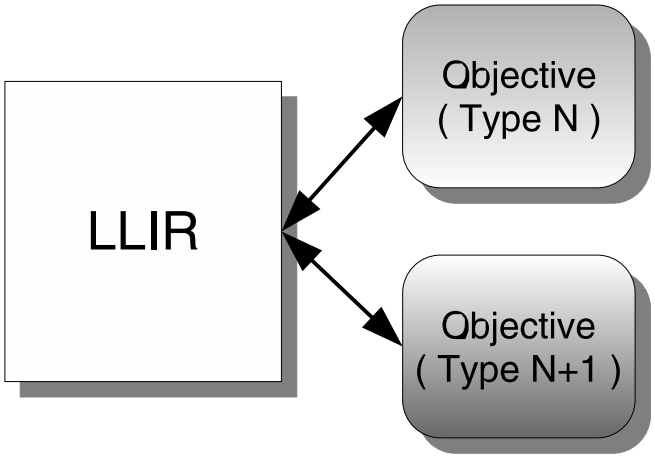


Figure 5.7: Concept of a single objective

The first idea was the design of a single overall objective for each involved application. It would be assigned to the entire LLIR representation which is the top of the hierarchy. At first view, this concept seemed to be the simplest and a sufficient solution. An objective would store all desired data and could be accessed by a function stored in the LLIR class (see figure 5.7). The main drawback arises from both the complicated storage and the retrieval of data. Since the information is not directly assigned to the LLIR components, all accesses are compelled to consult the LLIR root object. In the current version of the LLIR, only function objects possess a direct routine to retrieve its dedicated LLIR representation. Other elements are required to traverse the hierarchy stepwise to get the top element. For lower elements of the tree structure, it becomes a chore. For example, LLIR operations had to read its encompassing instruction, then the corresponding basic block followed by

its function, and finally the root entity. To avoid these multiple function calls, the LLIR elements could be extended by adequate functions. However, this step would violate the condition of minimal modifications to the LLIR code. Furthermore, this concept would significantly complicate the development of objectives. Each of them had to hold information on all potential LLIR elements. Thus, complex data structures and functions had to be provided. The emerging design and realization challenge of objectives would be several times greater than the challenge that comes up with the current improved concepts.

A way out of the dilemma mentioned above is presented by a improved approach which assigns individual objectives directly to the proper LLIR elements. Each information carrier accommodates merely data that is significant for its low-level representative. In this fashion, both the structure of the objectives and their interlinking are substantially simplified. The objectives jettison their complex data structures since there is no need to store all arising information depending on their LLIR elements. Moreover, the entire underlying mechanism to integrate objectives and to fetch their data increases the ease of use. The information is addressed directly by the LLIR components, and thus makes the cumbersome retrieval of the LLIR root entity redundant. However, one disadvantage still remains. Due to the fact that the objectives are directly assigned to LLIR components, each class has to be separately extended by suitable functions. Besides the undesirable amount of modifications to the LLIR code (see section 5.3), these extensions lead to an inconsistent interface. Different types of LLIR elements provide varying functions and cause a difficult handling of objectives. The final concept tackles the problem of the inefficient interface. The framework is extended by the concept of a handler and results in a productive collaboration of the LLIR and their objectives. Its design and realization is thoroughly described in the following.

Due to inheritance, each LLIR element is assigned an individual handler. It is implemented in the class LLIR_TaggedElement which serves as a base class for most of the other structural components. The general purpose of the handler is to provide a connection between particular objectives and its LLIR elements. Hence, a well-defined separation is accomplished which allows an independent development of both software modules. To obtain potential data that are placed in an objective, a two-step system is intended to be run through. Firstly, it must be checked if the desired information is available at all. After consulting the handler and receiving a positive response, the second step performs the actual read operation. The handler is involved again in order to establish a connection to the objective that exposes its data.

Each handler manages the individual objectives that are assigned to a particular LLIR component. In order to allow dynamic behavior, a list was chosen as data structure. It is not constrained by a predefined fixed size but may be arbitrarily expanded. Thus, each handler can hold any desired number of objectives. Besides, the handler class provides auxiliary functions which aid to steer the linking process between the LLIR and their objectives. The main functions are briefly introduced in the following paragraphs.

Each objective requires a unique identification in order to be distinguished by a handler.

This is important to verify if an individual handler is already assigned a specific objective. Moreover, this identification prevents to add multiple objectives of the same type to a single handler. For this purpose, each objective is assigned a unique type, e.g. WCET or BCET.

When a new objective is created, the corresponding handler must be notified. If the objective type is integrated into the framework for the very first time, the class attribute `ObjectiveType` must be extended by the type of the objective. After this adjustment, all functions can be employed for the new type class. In the next step, the individual handler of the particular LLIR object is retrieved using the function `getHandler`. Afterwards, its objective list is to be updated. For this purpose, the function `addObjective` is invoked. It expects the newly created objective as function argument and appends it to the handler list. The extension to the list succeeds only if there are no other objectives of the same type already managed by the handler. This constraint makes sense since a particular LLIR element may be assigned at most one objective of a specific type. For example, it is practically infeasible that a basic block possesses more than one worst-case execution time estimate for a single program. Hence, adding two WCET objectives to the handler of a single basic block would be trapped by the function `addObjective`. This function is notably involved in the converter crl2llir which transforms the results produced by the aiT WCET analyzer from the CRL2 representation to the LLIR.

The function `hasObjective` determines whether a given objective type is present in the current state of the list. As described above, it is guaranteed that each handler list contains at most one objective of a specific type. For this reason, each list element is unambiguously denoted by its type. For the sake of simplicity, the argument of function `hasObjective` is therefore an objective type which is specified by the enumeration type `ObjectiveType`. For example, if the user wants to know whether there is any worst-case execution time information available for a given basic block, he calls the function `hasObjective` with argument `WCET`. The return value is of type boolean and indicates if the search was successful.

As a counterpart to the function `addObjective`, `getObjective` is responsible for the retrieval of objectives which were added by the former function. In the same fashion as `hasObjective`, the function is invoked with an objective type as argument. This unique identifier allows a simplified handling. It is intended to call this function in conjunction with `hasObjective`. In this two-phase model, the first step ensures that the sought objective has been assigned to the LLIR element yet. If it is stored in the objective list of the handler, the fetch process can be continued by performing the actual function call.

The substantial benefit of the handler is the clear separation between the objective framework and the LLIR elements. Both parts can be developed independently since they do not rely on each other. Due to the incorporation of the handler constructs to the base class LLIR_TaggedElement, each newly created LLIR element is automatically equipped with the handler mechanism. Also, the addition of new objectives can be easily accomplished. Basically, the list of supported objectives, which is an attribute of the handler, has

to be slightly extended by the new newly added objective type. After these small modifications, the objectives are interlinked with the LLIR and can be afterwards utilized. A simplified overview is depicted in figure 5.8. The handler is managing three distinct objectives, where two of them are holding worst-case and best-case execution time data, respectively.

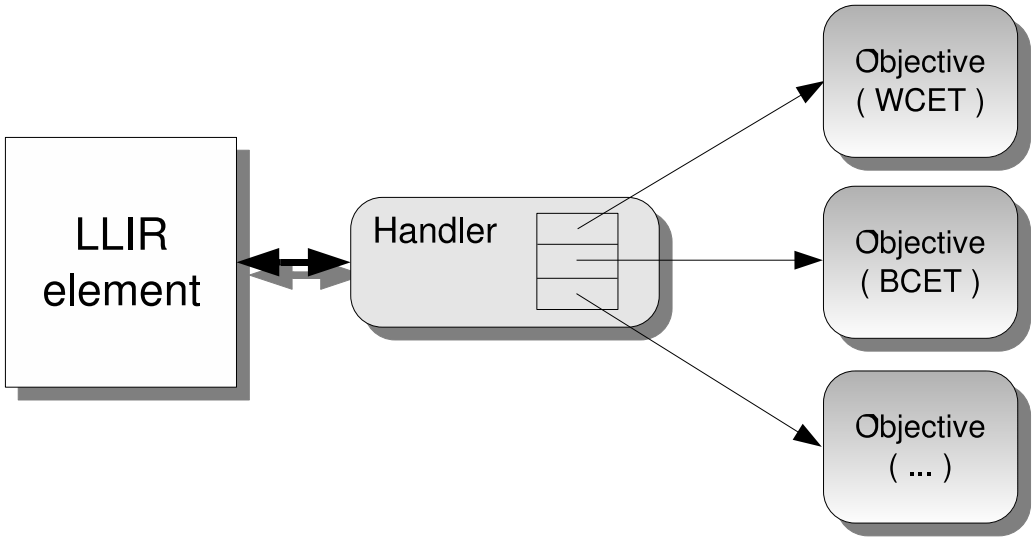


Figure 5.8: Handler mechanism

Objectives can be freely designed to suit users' requirements. They are not limited to any design structures or paradigms. However, there are few functionalities that must be available with any objective. These functions mainly serve the handler to enable a proper collaboration with the different objectives. In order to observe these constraints, the concept of *abstract classes* is applied. Abstract classes represent a generic or abstract concept from which more specific classes are derived. They can not be instantiated themselves and serve merely as a base class. To be referred to as an abstract class, the base class is compelled to encompass at least one *pure virtual function* which has mandatorily to be defined in all derived classes. To refer to the handler concept, the class `Objective` represents the abstract class, whereas the actual objective classes (like the worst-case execution time information carrier `WCET_OBJ`) are referred to as base classes. Each objective class must provide a mechanism to expose its type. This task is carried out by the member function `getObjectiveType`. Its return value is an objective type defined in the handler class. It is the same attribute that is required to be extended when a new objective type is intended to be supported by the handler. Thus, it serves as a unique identification. The aforementioned function is applied by the handler when new objectives are interlinked with the LLIR elements. More accurately, in order to avoid the assignment of multiple objectives of same type to an LLIR element, a check is performed. This procedure retrieves the type of the new objective and compares it with the already accommodated types. In the case of a correlation, adding of the objective is rejected.

Besides the specification of the desired functions, abstract classes aid the process of managing objectives of different types. This paradigm is called *polymorphism*. It enables to write programs which process objectives of classes that are part of the same class hierarchy in the same way as if they were all objects of the hierarchy's base class. This feature is exploited by the handler. On the one hand, all individual objectives of various types are held in the same data structure and are treated equally. On the other hand, they provide a broad spectrum of specialized functionalities.

Classes derived from the abstract class represent the actual objectives which are designed for a specific field of application. The user is free to tailor them to his requirements and is not required to meet any predefined constraints. There are only few specifications the objectives are expected to accord to. These requirements arise firstly from the nature of abstract classes; secondly, they emanate from the prerequisites of a unique objective identification. Each abstract class contains mandatorily at least one pure virtual function. This function must be defined in the derived classes. As mentioned previously, the handler mechanism relies on a function that determines the objective type. Due to the fact that this functionality must be supported, the function `getObjectiveType` is supposed to be available in each objective entity. Another requirement refers to the constructor of every objective class. The objective type is treated by the handler as a unique identification and ensures that at most one specific objective is interlinked with an LLIR element. For this reason, each newly created objective has to accommodate its type specification using its constructor which can be accessed by the handler.

There are no limits to the diversity of objectives. Depending on the considered issue, they may be purposively designed. Present-day compilers optimize code for various purposes. Most of them are employed on desktop systems by average consumers who translate source code written in a high-level language to a machine language. These types of systems do not rely on execution time bounds that must be met, but try to increase the average-case execution time (called *ACET*). Their compilers are not coerced into handling time specifications for their low-level components like basic blocks. A prominent example is the GNU gcc compiler [GCC05]. This versatile framework is developed by the Free Software Foundation and encompasses different frontends and backends for numerous high-level languages and machine languages, respectively. Also, the present version of the LLIR satisfies the compiler requirements for non-time critical desktop systems. All essential information is given in the form of a control-flow and data-flow graph, and therefore, there is no need for further supplementary objectives.

In contrast, real-time systems, which are part of embedded systems, depend on the observance of execution time bounds. The most common issue involves the worst-case execution time. It was the propelling problem statement which led to the present thesis. Besides the already discussed extensions to the LLIR and the handler framework, the first type of objectives, namely the WCET objectives, has been successfully designed and realized. A thorough description of their structure and their creation process will follow in section 5.5.

5.5 Conversion from CRL2 to LLIR

This section elaborates on the back conversion of the WCET analysis results from the CRL2 representation to the LLIR. The goal of this thesis is the extension of the current LLIR by WCET information. Hence, the compiler backend will be suited for the development of novel WCET compiler optimization algorithms as well as for the implementation of time-critical applications. Moreover, a high value has been set to the development of a homogeneous compiler framework. It allows one to exploit all available compiler analysis results which may be passed to the WCET analyzer in order to obtain best results.

As described previously, the LLIR has been successfully converted by llir2crl to an equivalent CRL2 representation. This CRL2 control-flow graph is supplied to AbsInt's WCET analyzer that performs its calculations and finally annotates the CRL2 graph with the sought attributes. Furthermore, adequate LLIR extensions and an interlinking mechanism between LLIR elements and the analyzer results have been realized. The remaining step is the retrieval of CRL2 annotations and their import into the LLIR.

This transformation process is carried out by the converter crl2llir. Unlike llir2crl, the converter uses the CRL2 graph hierarchy as basis. Basically, it walks through the graph in a top-down fashion. In the beginning, each CRL2 routine and its elements are scanned separately. The detection of valuable CRL2 annotations leads to the creation of new information carriers called objectives. They are assigned these annotations and are interlinked with the corresponding LLIR elements via handlers. The following paragraphs provide a simplified introduction to the converter and its proceeding, and describes the emerging problems in more detail.

The initial version of the CRL2 control-flow graph is constructed by the converter llir2crl. The converter reads the LLIR control-flow graph and constructs an equivalent CRL2 graph, i.e. there is a direct mapping between the elements of both intermediate representations. Each LLIR function and each basic block cause the creation of corresponding CRL2 routines and blocks, respectively. Besides, the tf14net library (see section 5.2.3) is utilized to generate equivalent CRL2 instructions and operations. Furthermore, each CRL2 block is annotated with an attribute which holds the label name of its LLIR counterpart. This identifier is used to achieve an unambiguous assignment of basic blocks accommodated in both representations. Finally, the LLIR basic block successors and predecessors are determined and deployed to model CRL2 graph edges.

The resembling control-flow graphs would allow a simple conversion of data between both intermediate representations. If the results produced by aiT were attached to the initial CRL2 graph, they could be easily forwarded to the LLIR elements. To do so, all CRL2 elements had to be traversed and their corresponding LLIR elements had to be retrieved by merely comparing their identifications. Last but not least, the LLIR components had to be equipped with the analyzer results.

However, aiT does not operate on the CRL2 file created by llir2crl. Before the inter-

mediate representation is supplied to the WCET analyzer, it is possibly modified by an auxiliary tool called crl2loop. This tool verifies if the CRL2 graph contains any loops. If so, the graph is modified in terms of the loop transformation.

5.5.1 Loop Transformation

Loops are of prime importance for the execution time of a program. It is well-known that most programs spend a large amount of time by iterating through loop bodies. Because of this reason, handling of loops deserves a special attention. Especially in the recent years, research in the domain of loop analysis has been propelled by modern speed-up processor features, e.g. caches, branch predictors or multi-issue pipelines. They substantially accelerate the execution of a program and are considered an inherent part of most processor architectures. However, they complicate the prediction of tight WCET bounds. Cache states rely on the execution history of a program, i.e. depending on the previous instructions, the cache content may vary. This circumstance yields cache hits and cache misses which tremendously affect the execution time. Loops are a prominent example which demonstrate the influence of caches. Iterations of the same loop, notably the first ones, indicate different execution properties. The very first loop execution encounters a cache miss and is compelled to fetch relevant data into the cache memory; this results in a relatively long execution time. Due to speculative prefetching, it still might happen that the cache content is modified while the second loop iteration is performed. For the next iterations, the situation stabilizes since the required data remains in the cache memory producing persistent cache access times. Hence, the beginning loop iterations encounter different cache contents than their successors. Due to this fact, a precise worst-case execution time estimation must take a distinction of loop iterations into account. Unlike aiT, some WCET analyzers treat the cache naively or even do not consider it at all. Their results are afflicted with a large overestimation and are of no use for hard real-time systems.

To enhance the WCET bounds, AbsInt's WCET analyzer examines various loop execution contexts. Originally, the loop code is located within a routine, i.e. the corresponding basic blocks that represent the loop are treated like other routine blocks. However, this structure does not enable a proper loop analysis. Thus, the control-flow graph is modified to enable a simpler handling. More accurately, each recognized loop is extracted into a separate CRL2 routine that can be assigned new contexts (see section 3.2) which in turn reflect the contrasting loop iterations. This approach is called *loop transformation* and is based on the *Virtual Inlining and Virtual Unrolling* algorithm [MAWF98].

An example clarifies the procedure of the loop transformation. Assume that this simple **`for`**-loop, nested in a routine, is given in a pseudo code:

```
routine R {
  BLOCK0;
  for ( BLOCK1 )
    BLOCK2;
```

```
    BLOCK3;
}
```

In the beginning of routine R, basic block BLOCK0 is executed. Subsequently, the loop header BLOCK1 is evaluated and possibly the loop body referred to as block BLOCK2 is executed. Before the routine is left, basic block BLOCK3 is interpreted. Figure 5.9 depicts the corresponding control flow. The solid arrows symbolize the regular control flow. The other arrows starting from the loop header (BLOCK1) denote the *true* and *false edge*, respectively. As long as the condition of BLOCK1 is met, the control flow repetitively executes BLOCK2 that is reached by the true edge. Otherwise, the false edge to BLOCK3 is traversed and the routine is left at the END block. One possible trace might be expressed in the following way:

BLOCK0 ⟶ BLOCK1 ⟶ BLOCK2 ⟶ BLOCK1 ⟶ BLOCK3.

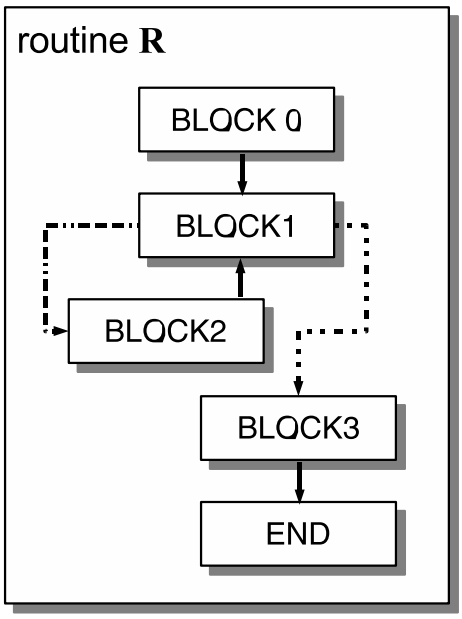


Figure 5.9: Control flow before loop transformation

This configuration complicates the data flow analysis (DFA) which is part of the WCET analysis. The former sets up equations for each node of the control-flow graph. These equations are obtained by calculating the output from the input and express dependencies between different data elements that are modified by the program. Referring to loops, the equation for the loop body is determined by combining the data flow value of the loop entry and the loop exit point. Due to the fact that loops may begin their iteration in a completely different state than encountered in further iterations, it is beneficial for the DFA to separate them from the routine. This modification of the control-flow graph is done by the loop transformation. The entire loop is shifted to a newly created CRL2 routine that is called from the original one. Furthermore, the loop iteration is substituted by recursion,

e.g. the loop calls itself before it returns to the loop header. In correspondence to the aforementioned code, the pseudo code after the loop transformation is defined as follows:

```
routine R {
  BLOCK 0;
  call_loop_R.L1: call R.L1;
  BLOCK3;
}

routine R.L1 {
  if ( BLOCK1 ) {
    BLOCK2;
  recursive_call_R.L1: call R.L1;
  }
}
```

The program has been split into two routines as can be seen in figure 5.10. The original routine `R` calls routine `R.L1` which represents the extracted loop. The routine name was chosen deliberately since it corresponds to the naming convention that is used by aiT. Loops in a routine are enumerated by decimal numbers, beginning with the decimal 1. Each routine that is created by the loop transformation is assigned a routine name that is a concatenated string of the original routine name, a colon and the enumerated loop L*n*. After evaluating the loop header `BLOCK1`, `BLOCK2` is executed and finally the loop calls itself recursively at `recursive_call_R.L1`. The recursion redirects the control flow back to the loop header. The *false* edge, which is traversed when the **`if`** condition is not met, leads the program through the `end` block back to the loop call node `call_loop_R.L1` in routine `R`, where the further program execution is resumed.

The presented loop transformation is supposed to provide a first notion of the modifications applied to CRL2 loops. Besides simply structured loops, most applications employ loops which contain multiple exits, i.e. the loop is not solely left at the end but may contain several **`return`** statements which serve as exit points. Consequently, the loop transformation slightly differs but the general concept remains [ait]. As a counterpart to the latter loop type, some few programs, notably library routines, operate on loops with multiple entry points. However, this loop type is rarely encountered and thus not supported by aiT.

Typically, the loop transformation is performed by the decoder exec2crl. After reading the binary program, the internal loop transformation modifies the internal control-flow graph, i.e. the original program is left unchanged. As mentioned before, the objective of the present master's thesis is to convert the control-flow graph, which is stored in the LLIR format, to CRL2. Therefore, exec2clr is not involved. However, the loop transformation is essential for a successful WCET analysis and is required to be performed subsequently. Once the initial CRL2 control-flow graph is available, it is passed to *crl2loop* which is an auxiliary stand-alone tool provided by aiT but is also incorporated in exec2clr. It extracts loops in the well-known fashion and generates an improved CFG for further analyses.

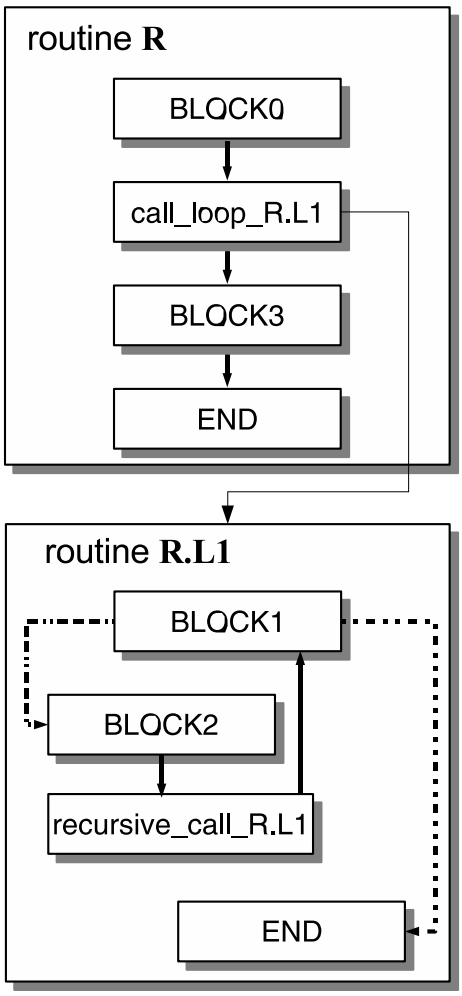


Figure 5.10: Loop extraction by the loop transformation

The loop transformation complicates the mapping of LLIR and their belonging CRL2 elements. Before passing the code to crl2loop, there is a direct dependency of objects in both representations. The converter llir2crl creates for each LLIR component an equivalent CRL2 object. The situation, however, changes as soon as the loop transformation moves CRL2 blocks to newly created routines. Firstly, the CRL2 *loop routines* have no corresponding LLIR functions; secondly, the shifted CRL2 basic blocks changed their location leading to a modified structure of the original routine.

A direct mapping of elements is essential for the converter crl2llir. It reads the CRL2 file generated by the aiT WCET analyzer and imports the annotated analysis results into the LLIR. More accurately, it extracts the results from the CRL2 elements and assigns them to the corresponding LLIR components. To enable this conversion, the last step expects a precise allocation of elements independent of previous transformations. Due to the fact that the results are exclusively stored in CRL2 routines and basic blocks, no mapping of

instructions and operations is required.

The absence of corresponding LLIR functions in terms of CRL2 loop routines arises the question where to store analysis results. Each CRL2 routine that was created by crl2loop is annotated with loop bound specifications. This information is of great importance and should be transformed into the LLIR. However, a direct assignment to an LLIR function is not possible. Therefore, an assumption is made. Each CRL2 loop routine is composed of CRL2 basic blocks that correspond to LLIR blocks. Thus, the sole reasonable connection between the modified CRL2 representation and the LLIR representation are the basic blocks representing the CRL2 loops. Hence, instead of assigning gained information from CRL2 loop routines to LLIR functions, the data is interlinked with the corresponding LLIR blocks. More precisely, all desired information of CRL2 routines is added to the appropriate LLIR block of the first CRL2 block representing this loop.

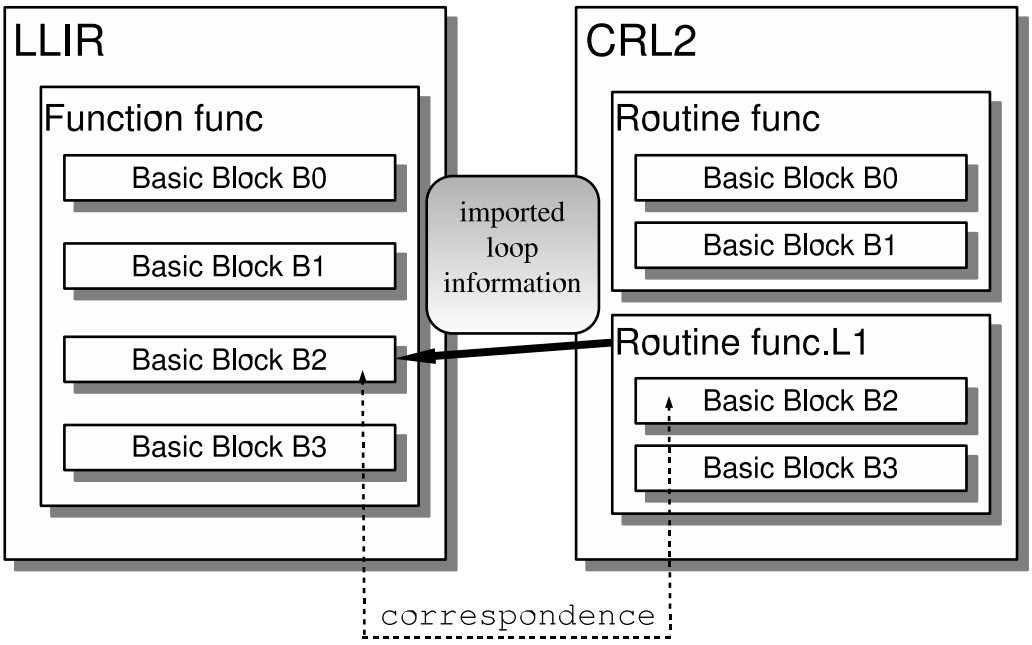


Figure 5.11: Assignment of CRL2 data to the corresponding LLIR block

This approach is depicted in figure 5.11. As can be seen, LLIR basic blocks `B2` and `B3` have been transformed to CRL2 basic blocks with the same indication. These blocks represent a loop. Therefore, they are moved to routine `func.L1` by the loop transformation. The analysis results concerning the loop bound specifications are assigned to the corresponding first block of the loop routine, namely the LLIR block `B2`. In this manner, valuable annotations of loop routines do not get lost but are added to the corresponding basic blocks by exploiting the relationship of blocks within the two intermediate representations. To retain consistency, data from regular CRL2 routines is assigned to the first LLIR basic block of the corresponding LLIR function.

The transformation of basic blocks by crl2loop provides another significant informa-

tion. It recognizes CRL2 basic blocks that represent a loop, and turns them into separate loop routines. Thus, due to the location of blocks, it can be inferred which LLIR blocks belong to a specific source code loop. These block annotations can be used to get rid of the tedious manual loop bound specifications. In the current state, AbsInt's aiT provides analyses for simply structured loops. Their bounds are automatically determined and employed to the corresponding CRL2 routines. However, the analysis of loops, which are normally implemented in everyday applications, often fails. In these cases, the user is compelled to provide these specifications manually in order to enable the WCET analysis at all.

This current state is not desired in a sophisticated compiler framework. The manual annotation process is tedious and error-prone, and incorrect annotations lead to wrong analyzer results. Thus, it is intended to develop an algorithm that analyzes any kind of loops and passes the loop bound specifications automatically to the WCET analyzer. Consequently, the knowledge of blocks which represent a loop is highly beneficial.

The converter crl2llir anticipates the design of the future loop algorithm and converts the information that indicates loop basic blocks into the LLIR. This data eases the development of the loop analyzer since there is no more need to detect dependencies between source code loops and their LLIR blocks. However, the realization of an efficient loop analyzer is a complex process and goes beyond the scope of this thesis. Therefore, its contribution is restricted to the provision of the supplementary block data.

It is not mandatory to find LLIR functions the CRL2 loop routines have been derived from since all CRL2 routine annotations are attached to LLIR blocks. However, knowing the origin of derived loop routines enables an efficient search of appropriate basic blocks. Instead of walking through all LLIR functions to find the counterpart of the CRL2 block, merely the proper routines that accommodated the blocks before the loop transformation are examined. This results in a substantially minimized complexity of the developed converters. The proceeding of crl2llir resembles the approach of the converter llir2crl. The program hierarchy is traversed top-down i.e. a system of nested loops begins at the top of the entire control-flow graph and reads then all CRL2 routines as well as their elements successively.

As mentioned above, a mapping of CRL2 routines and LLIR functions is required to allow an efficient search for corresponding basic blocks. Original CRL2 routines created by the converter llir2crl allow a simple retrieval of their LLIR counterpart. Since both objects bear the same name, a string comparison is sufficient. In contrast, CRL2 routines created by the loop transformer possess a name with no correspondence to any LLIR function. As mentioned previously, the routines created by the loop transformation observe a predefined naming. Each loop stored in the source code is enumerated with a decimal number beginning with the value 1. Even nested loops follow this rule. The outermost loop is denoted by 1, inner loops are designated by an incremented number. This decimal identifier is applied to generate the name of the routine created by crl2loop. Each loop routine derives its name from its surrounding function. In addition, the aforementioned identifier is used as suffix. To clarify the naming convention, consider this simple C code:

```
int main() {
  int i, j, k;
  for (i = 0; i < 10; ++i )
    for ( j = 0; j < 10; ++j )
      for ( k = 0; k < 20; ++k )
}
```

The basic blocks, which represent the three loops, are moved to three separate routines. The CRL2 block of the outermost loop using the integer variable `i` causes the creation of a CRL2 routine that is named `main.L1`. Here, the prefix `main` comes from the function name of the source code; the suffix `.L1` is a concatenated string including the identifier for the first loop. The CRL2 blocks of the second and third loop are placed in routines called `main.L2` and `main.L3`, respectively.

Knowing this convention enables an easy mapping of CRL2 routines and LLIR functions. Basically, the suffix has to be removed to obtain the original name of the routine. Hence, searching for LLIR blocks, which correspond to the CRL2 blocks moved by the loop transformation, is restricted to one LLIR function. This is notably advantageous for complex programs which are comprised of numerous functions and loops. Instead of comparing a large number of LLIR blocks with the corresponding CRL2 block, the search is limited in a targeted way to one sole LLIR function.

Besides the examination of routines, the mapping of CRL2 and LLIR basic blocks is mandatory. Basic blocks represent program units which hold numerous significant annotations. Especially, timing analyses (like the WCET or BCET estimates) produce results that are meant for blocks. For instance, AbsInt's WCET analyzer calculates WCET estimates and execution counts (number of block executions when critical path is traversed) which serve as basis for the calculation of the global WCET. Both values are attached to basic blocks.

Within an assembly file, basic blocks have unique labels. When the LLIR gains its data from this file, it also imports the labels. They are stored as LLIR block attributes and serve as their identification. Unlike the LLIR, CRL2 does not envision a unique name for its basic blocks. However, as discussed above, the converter of this thesis relies on an unambiguous mapping of blocks. Therefore, CRL2 blocks must be assigned the same unique labels as given for the LLIR blocks. To do so, the possibility of annotating CRL2 elements with arbitrary attributes is exploited. Whenever a new CRL2 block is derived from an LLIR block, the label of the latter is taken into account. In this manner, blocks can be always relocated even if they have been moved by the loop transformation.

By means of the described techniques, an efficient mapping of basic blocks belonging to the two intermediate representations is not an issue. Loop transformations, which increase the precision of analysis results, can be performed without any restrictions. Although the original hierarchical graph structure is modified, the well-known naming of CRL2 loop routines and the auxiliary identification attributes of CRL2 blocks allow an ef-

ficient retrieval of the appropriate LLIR elements. Hence, the back-conversion from CRL2 to LLIR can be easily performed.

After resolving the mapping problem spawned by the loop transformation, the main task of crl2llir is now presented. The detection of valuable aiT results in the form of CRL2 component attributes demands a transformation to the LLIR. As stated earlier, the gained data is not directly attached to the LLIR elements but is embedded in objectives and interlinked via handlers. Objectives represent information carriers which are designed to accommodate supplementary information. They are not bound to any conditions and may therefore be tailored to numerous applications. However, this thesis concentrates on the worst-case execution time and will restrict the subsequent descriptions to WCET objectives.

The converter crl2llir is the first application in the developed framework which makes extensive use of the objective-handler mechanism. The relevant information, which is desired to be imported to the LLIR, is assigned by aiT to CRL2 routines, CRL2 blocks and CRL2 edges. The lower elements of the CRL2 hierarchy, namely instructions and operations, are not observed since they do not provide any sought analyzer results. Like the first converter of the developed toolchain, crl2llir utilizes nested loops to walk through the entire graph hierarchy.

The top-down approach begins with CRL2 routines which are examined successively. For each routine, its basic blocks are read. Whenever a valuable annotation is found, a new WCET objective is created. In the first step, the converter calls the appropriate objective function and passes the read annotation as argument in order to store it within the internal objective data structures. In the subsequent step, the corresponding LLIR element is determined by means of the techniques described above. Next, the full strength of the handler concept is exploited. Since the supplementary CRL2 results are not directly assigned to the LLIR elements but use objectives, the converter is not compelled to invoke element-dependent functions. It does not make any difference whether data, for example, is attached to the LLIR graph directly or to single basic blocks. Instead, the converter retrieves the individual LLIR element handler by using a standardized function (`getHandler`) and passes the WCET objective as function argument. In this simple manner, the objectives are interlinked with the appropriate LLIR elements.

After finishing the transformations, a fully annotated LLIR intermediate representation is available. It accommodates all essential WCET information that enable to draw conclusions about the execution time of the represented program. Furthermore, details on existing loops are provided. Together with the results stemming from the previous compiler analyses and optimizations, this homogeneous framework opens up new vistas, e.g. the development of novel algorithms or real-time applications.

Figure 5.12 indicates the resulting annotated LLIR structure. As can be inferred from the diagram, the original LLIR representation is composed of one function called `func`. The latter contains three basic blocks. The right hand side of the figure shows the corre-

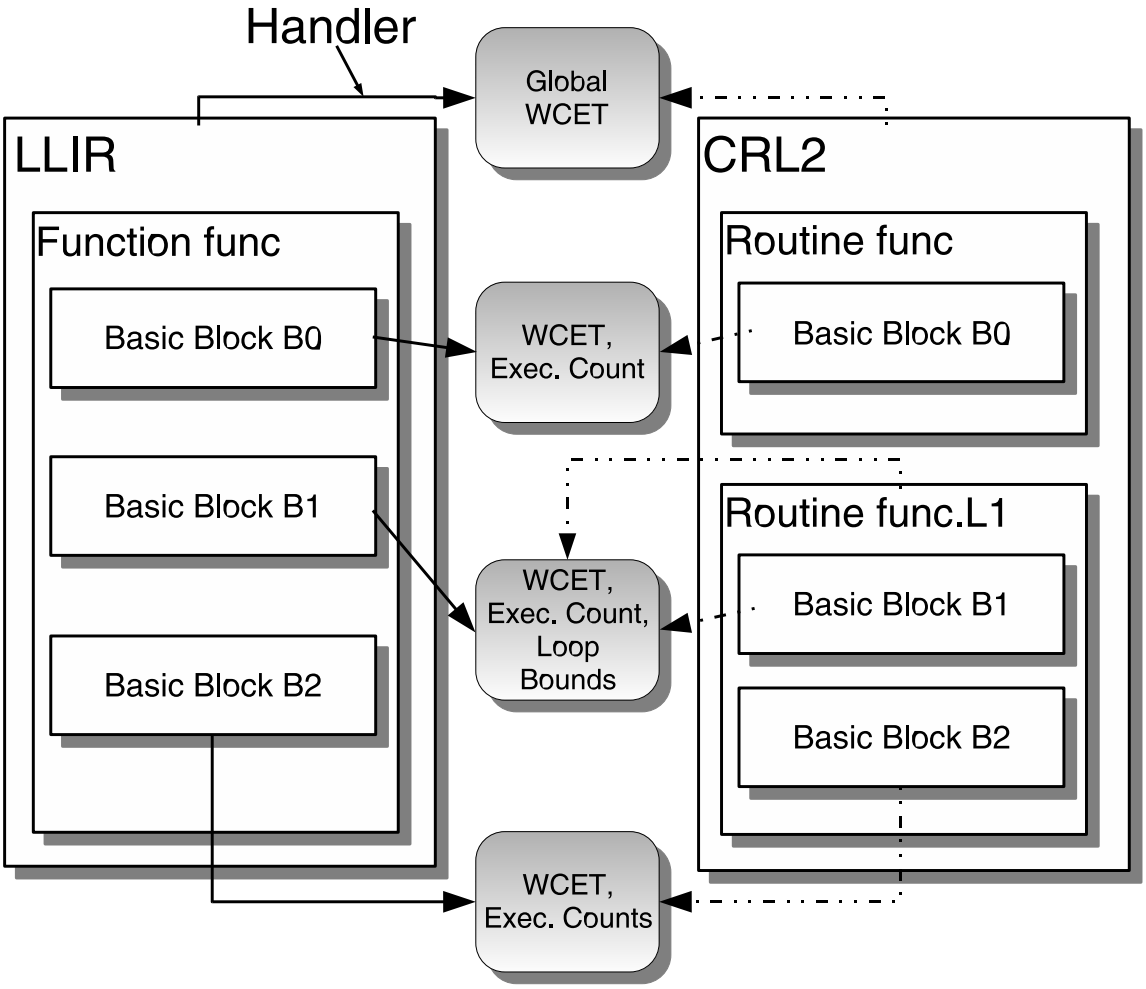


Figure 5.12: LLIR annotated with supplementary information

sponding CRL2 representation which has already been modified by the loop transformation. CRL2 basic blocks `B1` and `B2`, which form a loop, have been moved to a new CRL2 routine called `func.L1`. The ball-shaped entities represent individual WCET objectives. They are derived from the CRL2 elements and are interlinked with LLIR elements via individual handlers. As an example, three different types of WCET objectives are depicted. The objective at the top of the figure is attached to the entire LLIR graph and holds the global worst-case execution time of the entire program. The second and the last objective are basic information carriers which provide the WCET and the number of execution counts for a particular basic block. Finally, the objective bound to the LLIR block `B1` serves two purposes; firstly, it stores the regular block values, viz. the WCET and the execution count. Secondly, the objective holds loop bound specifications which have been derived from the CRL2 routine `func.L1`. This solution of assigning CRL2 routine information to LLIR basic blocks is mandatory since there is no corresponding LLIR function which would enable a direct data exchange between the latter and the CRL2 routine.

The current version of crl2llir supports the conversion of the following analyzer results:

- **Global WCET** for the entire program
- **Contexts**
- **Context-dependent execution counts** for single basic blocks
- **Context-dependent WCETs** of single basic blocks
- **Designation of *loop blocks***
- **Loop bound specifications**

The enumerated objects as well as the way they are stored in the WCET objectives will be introduced in the following.

5.5.2 Global WCET

The global worst-case execution time is the result of the path analysis which is part of the aiT WCET analyzer toolchain. Its calculation, in turn, is based upon the results of the pipeline analysis. To obtain precise WCET estimates, the aiT analyzer assigns its results for the CRL2 basic blocks not directly to the blocks but to their control-flow edges which serve as connections between the blocks. The pipeline analysis determines $T(e,c)$ that designates the worst-case execution time for a specific edge e and context c. In addition, aiT annotates each control-flow edge with the execution count $C(e,c)$. This value expresses the number of control passes along edge e and context c.

To obtain the context-free WCET for a specific flow of the program, the execution count for each involved edge e and context c must be known. The upper time bound for this program execution is calculated by accumulating the products $C(e,c) \cdot T(e,c)$ over all pairs (e,c). The evaluation of the global WCET for the entire program is an extension of the previous problem. Instead of taking all edge-context pairs of a specific program run into account, all feasible execution counts depending on their edges and contexts are involved. To obtain the overall worst-case execution time, the longest execution path of the program, which may ever occur, has to be found. To do so, all possible paths expressed by the accumulated products of the worst-case execution time t_{edge} and the execution count c_{edge} have to be analyzed. Finally, the maximal sum must be determined:

$$global\ WCET = \max(\sum_{\forall edge} t_{edge} \cdot c_{edge})$$

Due to the nature of this optimization problem, a linear constrain system is set up and the objective function is solved using integer linear programming (ILP) [Sch86].

This global WCET is assigned by aiT as a global attribute to the CRL2 control-flow graph. The converter crl2llir reads this value, creates a new WCET objective, and calls

the objective function `setWCET` with the WCET estimate as function argument. In the next step, the handler of the LLIR graph is retrieved and the previously created objective is handed. In this manner, the LLIR is extended by the worst-case execution time of the program it represents.

This extension can be used for both the development of real-time systems and as a helpful identifier during the development process of an optimizing algorithm. In the first case, developers can prove if the hardware that will be used as platform for the software is sufficient to meet all time constraints. This verification is highly important since the violation of these constraints may lead to outright disasters. If the constraints can not be met, either the hardware resources have to be increased or the complexity of the software must be reduced. On the other hand, the global WCET may be used as an identifier to measure the performance of new optimizing algorithms which aim at the reduction of the WCET. New ideas and approaches can be involved and their impact may be evaluated with this global identifier.

5.5.3 Contexts

WCET analyzers benefit from the use of contexts. Taking contexts into account, enables a more subtle analysis and this results in enhanced worst-case execution estimates. To be more precise, two parts of the timing analysis process, namely the value and the path analysis, exploit contexts.

The goal of the value analysis is to determine possible values in the processor registers for any possible program point. Without contexts, maximal value ranges have to be assumed in order to satisfy all potential program executions. Often, this leads to substantially oversized ranges that do not reflect the actual program behavior and therefore falsify the analysis results. Contexts tackle this problem by refining the value ranges. Instead of considering ranges of maximal size, they are split into subtle portions which are restricted by real values that occur for a particular point of the program. These ranges are assigned individual contexts and can be analyzed separately. The result is an enhanced value analysis.

The path analysis pursues the goal of calculating the global WCET by finding the longest feasible execution path. Its calculation is notably based on the execution counts and the worst-case execution times of the blocks. In the case of omitted contexts, path branches can not be analyzed properly. Due to the fact that one single context is considered, exclusively the worst case must be assumed in order not to violate the WCET safeness criterion. Consequently, this yields less tight time bounds. Like for the value analysis, the adoption of contexts offers the opportunity to assign individual contexts for each branch. Thus, a more precise path examination is possible and this finally leads to improved analysis results. The aforementioned paragraphs present by no means a complete description of contexts. They merely serve the purpose of an introducing repetition.

A thorough context specification is given in section 3.2.

While the aiT toolchain operates on the CRL2 file, it expands the intermediate representation stepwise by new contexts. A context is stored in the form of a call string that is a sequence of calls starting from the call in the entry routine. More accurately, call strings are comprised of subsequent terminals which represent a pair of a CRL2 basic block and a CRL2 routine. These terminal blocks contain a call instruction which directs the program flow to the corresponding terminal routine so that the following call string structure emerges: $bX \longrightarrow rX, bY \longrightarrow rY \ldots$ (b and r designate blocks and routines, respectively). To clarify the structure of contexts, consider figure 5.13, which represents a simplified CRL2 control-flow graph, as an example.

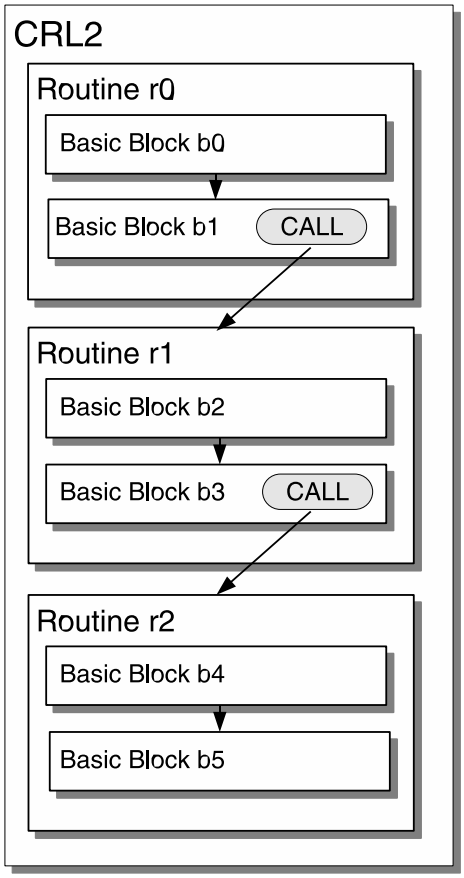


Figure 5.13: Representation of CRL2 contexts

Each of the CRL2 routines contains a basic block which in turn holds a `CALL` instruction. The `CALL` instruction in block `b1` performs a jump to routine `r1`. The latter accommodates basic block `b3` which uses its call instruction to redirect the program flow to routine `r2`. The resulting context call string is composed of two terminals and looks as follows:

$$b1 \longrightarrow r1, b3 \longrightarrow r2$$

Contexts are essential for a proper statement of the results generated by aiT. The two basic specifiers used for the estimation of the global WCET, namely the WCET and the execution counts of a basic block, rely on contexts. To point out the context issue, consider the storage of context-dependent values within the CRL2 as example:

```
path_cycles<c2>=5, wce_count<c2>=6
```

This exemplary extract is assigned to the control-flow edge between two CRL2 basic blocks and defines that the execution count for context `c2` denoted by `path_cycles<c2>` constitutes `5` control passes. Furthermore, the corresponding WCET estimate for one control pass in the same context indicated by `wce_count<c2>` is `6` execution cycles.

Needless to say that the maintenance of contexts is inevitably significant for the LLIR elements. In case of the absence of contexts, the aforementioned context-dependent results can not be stored in a sensible manner. They would occur in the LLIR without any reasonable dependencies, i.e. there would be no unambiguous assignment between the pair elements executions count and WCET estimation (see previous example). To meet these requirements, this thesis presents concepts and realizations which enable an elaborate integration of contexts into the LLIR.

Routines may be called with different arguments. This leads to varying initial states of the hardware, e.g. the content of processor registers or caches. These individual states are represented by contexts which are consequently assigned to CRL2 routines. If the CRL2 structure was not modified by the loop transformation due to additional routines, it would make sense to attach the context information to the corresponding LLIR functions. However, as mentioned previously, a direct mapping between both intermediate representations is not always feasible. Newly created CRL2 loop routines do not possess adequate LLIR functions. Therefore, a solution to this problem is the assignment of CRL2 routine data to the corresponding LLIR elements.

For the purpose of representing contexts within the LLIR, a new class has been designed. Like CRL2, the developed context class allows the storage of call strings. As introduced above, CRL2 call strings are represented by a sequence of block-routine pairs. To achieve the same structure, the new context class utilizes a lists of pairs as internal data structures. Each pair holds an LLIR basic block and an LLIR function as first and second element, respectively. To obtain a sequence of block-function pairs, these pairs are stored in a list. Due to this dynamic data structure, the call string list can be arbitrarily extended. Furthermore, each context is assigned a unique identification. As can be seen in the aforementioned CRL2 example, this identification is employed to enable an unambiguous representation of context-dependent aiT results. To support an adequate identification, the context class implemented in this thesis provides an attribute that can be assigned a unique specifier. Figure 5.14 gives a pictorial view of the developed class structure.

The integration of contexts takes place in two phases: the translation of available CRL2 contexts into the LLIR and the interlinking with context-dependent attributes. In an ini-

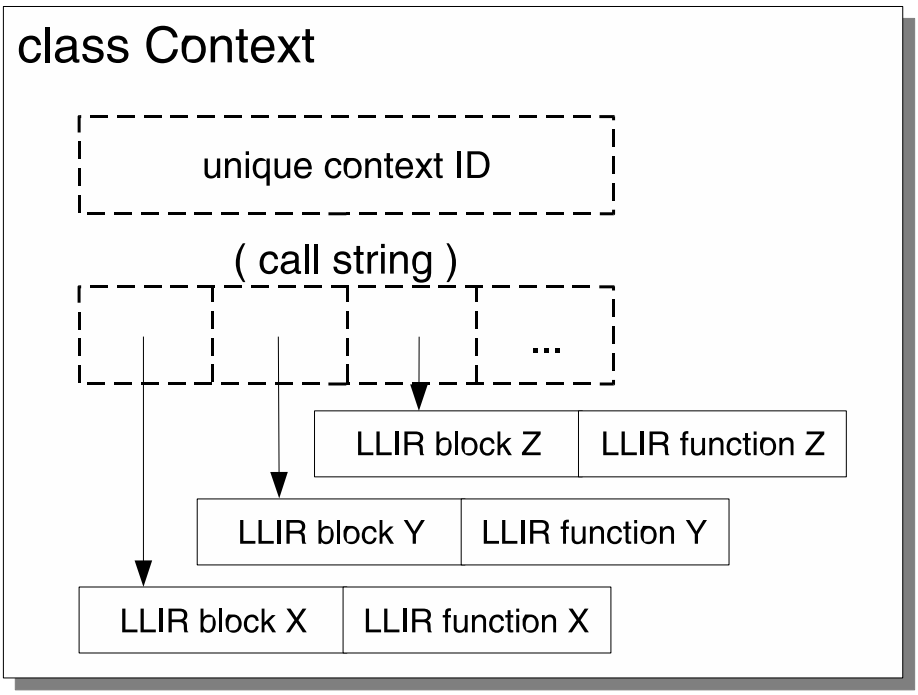


Figure 5.14: Overview of the class structure representing an LLIR context

tialization phase of the converter crl2llir, the entire CRL2 representation, which has been annotated with analysis results by AbsInt's WCET analyzer, is scanned for contexts. To do so, each CRL2 routine is checked. When a CRL2 context is found, the converter crl2llir creates a new LLIR context. Subsequently, the CRL2 context is analyzed, i.e. each block-routine pair is considered separately and its elements are extracted. After determining the corresponding LLIR block and LLIR function, these two objects are passed to the WCET objective function `addContextTerminal`. Finally, the ID of the CRL2 context is determined and used as argument for the call of the WCET objective function `setContextID`. As a result, an LLIR context object, which reflects the corresponding CRL2 context with its call string and ID, is available. In the next step, a new WCET objective is created. It is passed the previously generated LLIR context by calling the WCET objective function `appendContext`. In the case of multiple CRL2 contexts assigned to a CRL2 routine, the sequence of steps to create an LLIR context is repeated and each of them is added the the WCET objective.

The final step of the first phase is the interlinking of the created WCET objective to the appropriate LLIR element. For well-known reasons, the WCET objective is not assigned to an LLIR function but to an LLIR block. First, the first CRL2 block of the considered CRL2 routine is determined. Second, its corresponding counterpart, an LLIR block, is searched. Finally, the individual handler of the LLIR block is exploited to interlink the context information, which is embedded in the WCET objective, to the LLIR. At the end of the initialization phase of the converter, each of the LLIR blocks that correspond to the

very first CRL2 block of a CRL2 routine is annotated with context information (see figure 5.15). Hence, a basis for further steps is provided.

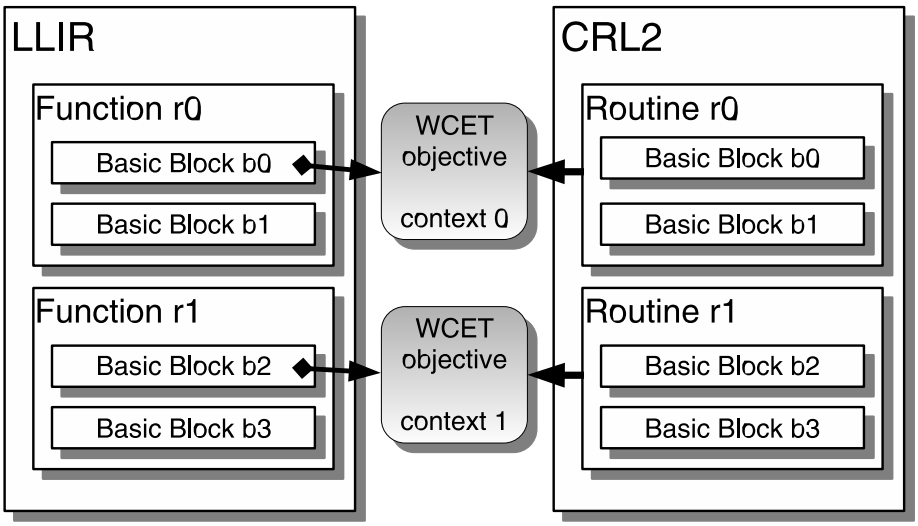


Figure 5.15: Assignment of context information

For the sake of efficiency, context data is not attached to each LLIR basic block of a routine but merely to the first one. The remaining LLIR blocks can yet call the function `getContexts` of their WCET objective to retrieve a list of available contexts. The mechanism behind this function verifies if the calling objective contains a non-empty context list. If so, the list is returned. Otherwise, a further distinction between basic blocks that represent a loop and non-loop blocks is required. In the first case, all basic blocks located in the same loop as the calling block must be identified. This feature is provided by examining the WCET objective function `getLoopPosition`. From this set of LLIR blocks, the first one is determined since it accommodates the context list. The calling basic block consults the first loop block and obtains in this manner the sought list of contexts. Otherwise, non-loop blocks demand the first block of the LLIR function they are part of and request its context list.

After the initialization phase, the converter crl2llir proceeds with the second traversal of the CRL2 control-flow graph. This time, the well-known approach of obtaining aiT results and interlinking them via handlers is performed. Besides the context-free LLIR extensions like the loop bounds and the designation of loop blocks, the attributes *execution count* and *block WCET* expect the specification of a context. Before they are added to the LLIR block, the converter consults the context list to receive the corresponding element. Thus, both the aiT result and the belonging context are attached to the LLIR basic block. The description of handling context-dependent execution counts and WCETs is presented in the following sections.

5.5.4 Execution Counts

The specifier *execution count* indicates the number of control passes between two basic blocks. To obtain more precise WCET estimates, the indicator is coupled to control-flow edges between two blocks. Moreover, an enhancement of the WCET analysis is achieved when contexts are taken into account. Therefore, each execution count attribute is specified by both the corresponding edge and context. Expressed as a function of edge `e` and a context `c`, it is defined as follows: `E(e,c)`.

Execution counts are employed to calculate the global worst-case execution time of an entire program. Together with the block WCET, both values are used by the path analysis to find the longest execution path. For some particular cases, namely for loops, the total number of execution counts for a limited set of basic blocks may be defined by user annotations. AbsInt's WCET analyzer as well as the developed converter llir2crl may be supplied with loop bound specifications. These values are retrieved by the loop transformation which turns the blocks that represent a loop into separate routines. Obviously, the loop bound specifications steer the number of loop iterations, and therefore correspond to the accumulated execution counts for this block set.

aiT determines execution counts during the pipeline analysis and assigns the calculated values to CRL2 edges. These context-dependent annotations occur in the CRL2 representation in two forms: `pipe_cycles<cX>=Y` and `pipe_impasse<cX>=Y` (`Y` and `X` designate a decimal number). The former declares `Y` execution counts for the corresponding CRL2 edge in context `cX`. The second representation signalizes that a path traversal through this edge in context `cX` is not feasible and must not be involved in further calculations.

The converter crl2llir carries out the transformation of CRL2 execution counts to the LLIR. A direct assignment, however, may not be performed since the LLIR does not exhibit control-flow edges to the user. The LLIR CFG is based upon a graph data structure but does not permit direct access to its edges. Instead, the edges can be retrieved indirectly by determining the successors and predecessors of a basic block. For this reason, execution counts are added to the LLIR blocks that correspond to the CRL2 blocks which serve as starting point for the CRL2 edge.

In order to extract the sought CRL2 annotations, each CRL2 block as well as its edges are examined. CRL2 basically distinguishes between two types of edges, namely *TRUE edges* and *FALSE edges*. They are applied to model control-flow branching points. Ordinary edges between two basic blocks, which do not rely on any conditions, are treated in the same manner as TRUE edges. A prominent example for different edge types is the conditional **`if`**-statement of high-level languages which causes the program either to execute the subsequent **`if`**-block or to skip it. Another example are loops which iterate through the loop body as long as the loop header conditions are met. On the low-level representation, the loop body is repetitively entered by the TRUE edge and finally skipped by the FALSE edge. Due to the absence of LLIR edges, a sensible storage of edge-dependent execution

counts demands a distinction of LLIR block annotations.

This requirement is met by providing corresponding WCET objective functions. When a non-zero CRL2 execution count annotation is detected, the type of its belonging control-flow edge as well as its CRL2 context are determined. Afterwards, the converter verifies if the corresponding LLIR block is not already interlinked with a WCET objective (created during the transformation of CRL2 contexts to the LLIR, see section 5.5.3). If not, a second objective is created, otherwise the existing one is selected[1]. Relying on the edge type, either the objective function `setExeuctionCountTRUE` or `setExecutionCountFALSE` is called. By passing the numerical value of the CRL2 execution count and the corresponding LLIR context as function arguments, the WCET objective is extended by this context-dependent attribute. This means that some particular LLIR blocks are interlinked with WCET objectives which accommodate both a TRUE and and FALSE execution count for a single context.

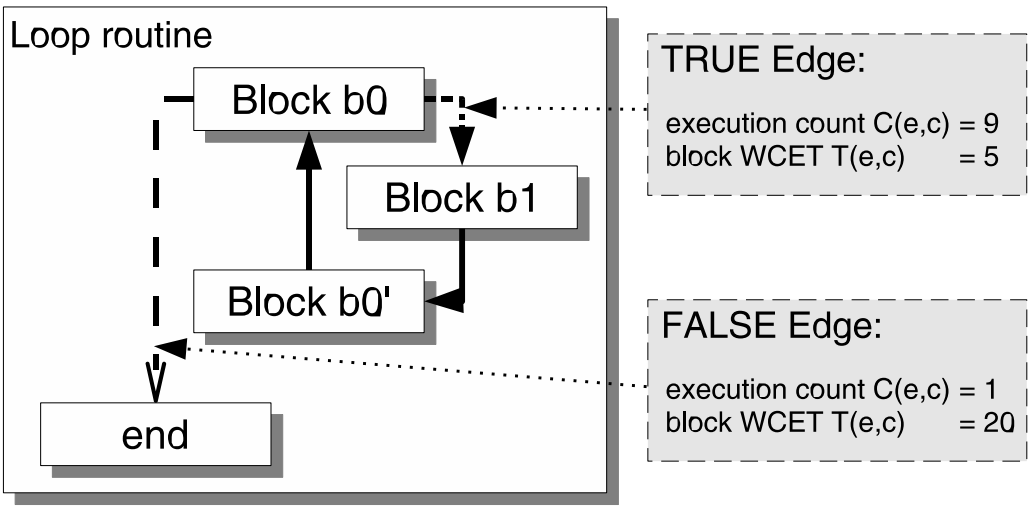


Figure 5.16: Mandatory distinction between TRUE and FALSE control edges

Figure 5.16 illustrates the necessity of distinguishing between both types of CRL2 edges. As can be seen, the figure depicts a CRL2 routine created by the loop transformation (see section 5.5.1). The loop is iterated ten times in total; nine loop iterations traverse the TRUE edge, the last one uses the FALSE edge to exit the loop. Besides the varying execution counts, both edges take a different amount of time to execute, namely 5 and 20 execution cycles for a single control flow, respectively. To reflect the analysis results in a reasonable fashion, the information of both edges may obviously not be merged. The sole solution is the storage of both values in separate WCET objective attributes.

[1]It should be emphasized that it is of great importance to operate on already existing WCET objectives. If a new objective was created and attempted to add to the LLIR block handler, the underlying mechanism would prohibit this operation, since a multiple assignment of objectives of the same type is not permitted.

5.5.5 WCET of Basic Blocks

aiT results concerning the worst-case execution time of single basic blocks resembles the CRL2 execution count annotations. AbsInt's WCET analyzer calculates the WCET for a single execution of a basic block in a specific context. The WCET can be expressed as a function which relies on the edge and the context: `T(e,c)`. Together with the corresponding context-dependent execution counts, Integer Linear Programming (ILP) is applied to determine the longest execution path which is indicated by the global WCET.

The WCET estimates generated in the final phase of aiT are again attached to CRL2 edges. They are designated by the well-known syntactical notation `wce_count<cX>=Y` that defines a time bound of `Y` execution cycles for a certain edge traversed in context `cX`.

The converter crl2llir handles the WCET estimates akin to the execution counts. For each CRL2 block, its control-flow edges are inspected. Due to different edge types, the requirement of accommodating edge-dependent WCET estimates within the LLIR emerges. For this purpose, the WCET objective class provides two functions, namely `setBlockWCETTrue` and `setBlockWCETFalse`, which enable the storage of context-dependent time bounds. Before adding the CRL2 annotations, crl2llir proves if the corresponding LLIR block is not already assigned a WCET objective that holds context information. Depending on the result, either a new WCET objective is generated or the converter proceeds with the already existing one. Finally, the corresponding LLIR context is determined and passed together with the actual WCET value to the objective.

In addition to the edge WCET estimates, objectives hold the overall basic block WCET. This indicator is not directly provided by aiT, and thus needs to be calculated seperately. While a CRL2 block is analyzed by crl2llir, the gained execution counts and WCETs are buffered. More accurately, for all available outgoing CRL2 edges of a particular basic block, the products of the execution counts and the edge WCETs sorted by contexts are temporarily stored and finally summed up. This accumulated value represents the upper time bound that might occur when this basic block is executed. Since this value indirectly includes all available contexts, it is stored as a context-free LLIR extension. The storage process is performed by invoking the WCET objective function `setBlockWCET` which merely expects a decimal value as argument.

Figure 5.17 illustrates the resulting situation after the converter crl2llir performed its transformations. For the sake of simplicity, both the LLIR and the CRL2 are composed of one function/routine which, in turn, contains three basic blocks. To store both CRL2 contexts, the converter generates two equivalent LLIR context objects. They are added to the WCET objective of the corresponding first LLIR basic block. The context-dependent annotations, namely the execution count and the single block WCET estimate, are retrieved from the CRL2 control-flow edge between `Basic Block 0` and `Basic Block 1` and are added together with the already present context `c1` to the `WCET Objective 1`. The latter is finally interlinked via a handler with the LLIR `Basic Block 0`. Additionally, the figure indicates the efficient storage of contexts. As mentioned previously, context

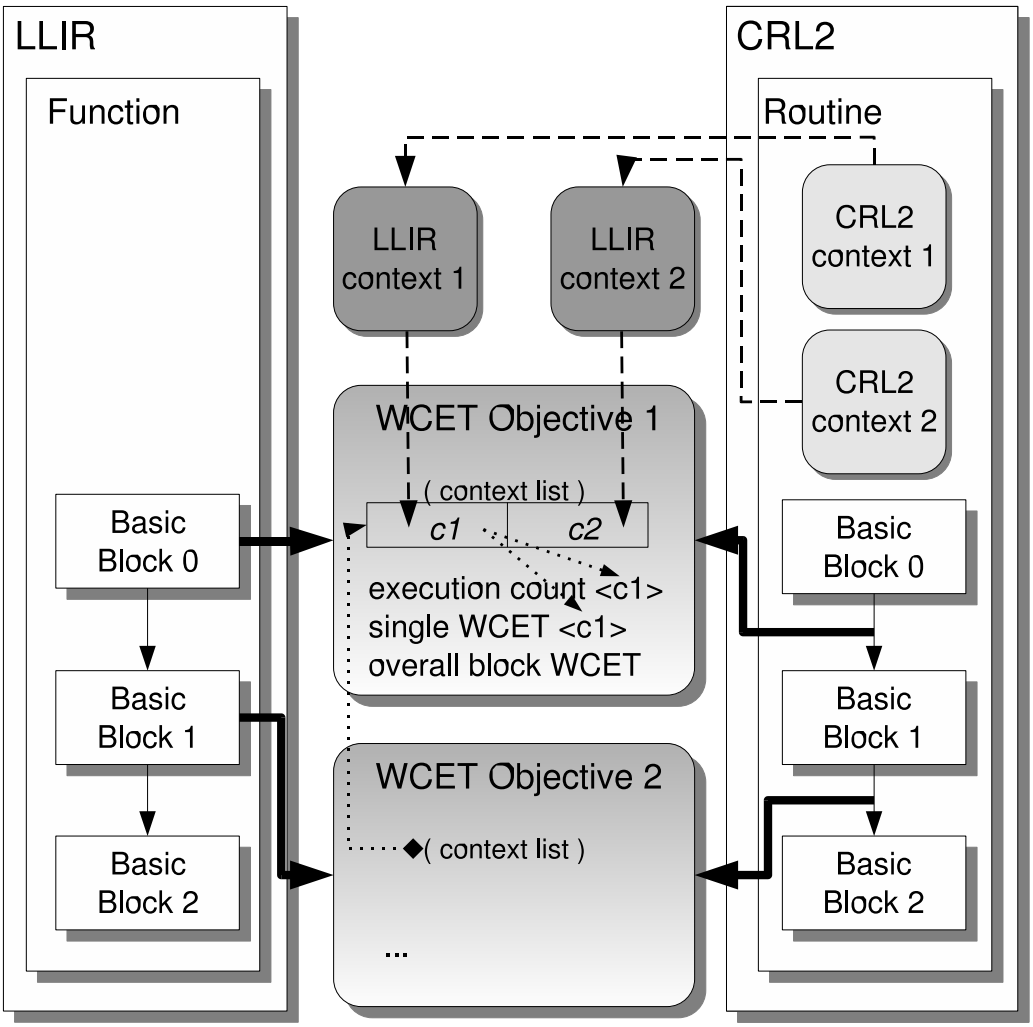


Figure 5.17: Context-dependent WCET objective framework

information is exclusively held in the corresponding first block of a CRL2 routine. Therefore, the context list of `WCET Objective 2` does not accommodate any elements. In order to obtain the required contexts, the objective is compelled to consult the context list of `WCET Objective 1`.

The LLIR extensions discussed so far suffice to provide a WCET-aware LLIR representation. However, the aiT toolchain generates further beneficial annotations which notably concern loops. Although the transformation of these extensions neither presents the prime goal of this thesis nor is directly advantageous for the current state of the developed WCET framework, it is of great importance for the future work. To enhance the quality of the compiler framework, the manual loop bound annotations have to be substituted by automatic mechanisms. To accomplish this goal, a sophisticated loop analyzer must be developed. The following LLIR extensions substantially aid the process of realizing this project.

5.5.6 Designation of Loop Blocks

The current state of the LLIR does not provide any approaches to analyze loops at the low-level. The converter llir2crl generates an equivalent CRL2 representation with an identical structure where each LLIR function and each LLIR block has a corresponding CRL2 routine and CRL2 block, respectively. For well-known reasons, the CRL2 graph is modified afterwards. Blocks that represent loops are shifted to newly created routines. The naming of the latter follows a well-defined convention: aiT enumerates each loop in the source code beginning with the decimal number 1 (see example in section 5.5.1). Thus, by analyzing the CRL2 routine name, it can be inferred which source code loop is addressed.

This fact is exploited by the converter crl2llir. Whenever a CRL2 loop routine is detected, its name is tokenized in order to extract the loop number. For each CRL2 block that is part of this routine, the corresponding LLIR block is determined. Afterwards, the converter retrieves the handler of the LLIR block and checks if it already possesses a WCET objective. If not, a new objective is created, otherwise the present one is reused. By using the objective function `setLoopPosition`, the loop basic blocks are annotated with the loop number.

This extension to the LLIR blocks leads to a clear classification of blocks that are involved in a loop and those that are not. By default, the WCET objective constructor sets the object attribute, which indicates the associated loop, to the value 0. When a CRL2 basic block has not been shifted by the loop transformation, the aforementioned attribute remains unchanged. Otherwise, it is assigned the corresponding non-zero loop number. After the converter crl2llir finishes its conversion, the WCET objective function `getLoopPosition` can be employed to retrieve the belonging loop number. LLIR basic blocks with the value 0 indicate that they are not involved in any loop. The remaining blocks, on the other hand, identify the loop they partially form.

The transformation of the loop membership can be considered as a *preparatory work* with the goal of achieving an annotation-free compiler framework. In the current state, neither the LLIR nor aiT have the ability to analyze loop automatically and to provide their results to other parts of the toolchain. For this reason, the user is compelled to supply the loop bound specification manually. This approach is highly error-prone and may lead to incorrect analysis results, let alone the tedious chore. The annotation problem is well-known and often subject of criticism [BH03, KP05]. However, no sufficient solutions could be presented yet. Most available WCET analyzers fail when it comes to the determination of loop bound specifications of ordinary loops that are employed in everyday real-time applications.

Due to the fact that the development of sophisticated loop analyzers is a highly complex issue, its development is beyond the scope of this thesis. It is, however, intended to begin with the construction of a loop analyzer in the near future. The already presented designation of LLIR basic blocks provides a basis for this intention: it simplifies the investigation of loops substantially since the involved basic blocks are already known.

5.5.7 Loop Bounds

The transformation of loop bound specifications resembles the designation of loop blocks. Due to a missing loop analyzer, the loop transformation reads the annotations which have been derived from the configuration file *wcetrc* (see section 5.2.1) and which were added by llir2crl to the corresponding initial CRL2 routine. The configuration defines local and global loop bound specifications including the names of the source code functions. When the converter creates a new CRL2 routine from an LLIR function, it checks whether there is a loop bound specification which is assigned the same function name. If so, the loop bound specifiers are attached as CRL2 routine attributes. These specifications are mandatory since their absence prevents the WCET analyzer from proceeding.

When aiT is analyzing the initial CRL2 representation, the loop transformation retrieves these annotations and transforms them together with the CRL2 blocks to the new loop routines. The back converter crl2llir recognizes these routines and their loop bound specifications. However, it can not transform any data directly to an LLIR function, since there is no corresponding function available. As mentioned in section 5.5.1, a way out of this dilemma is the assignment of loop annotations to LLIR basic blocks. At the beginning, the first CRL2 block of the loop routine is sought and its corresponding LLIR block is determined. Next, crl2llir retrieves the handler of the LLIR block and verifies whether it already contains a WCET objective. If not, a new objective is created, otherwise the converter reuses the existing one.

The loop bound specifications are imported into the WCET objective by calling its functions `setLoopBoundMin` and `setLoopBoundMax`. Figure 5.12 depicts a corresponding example for basic block `B1`. Its objective accommodates the WCET estimation, the execution count and the information on the CRL2 loop routine `func.L1`.

In the current state, the presence of loop bound specifications is not directly advantageous for the LLIR. The specifications are merely read from the configuration file and do not represent any analysis results. However, the data structures of the WCET objectives that accommodate the loop data are intended to be reused in the future. As mentioned previously, the development of a loop analyzer would significantly improve the quality of the entire WCET compiler framework. The system would get rid of manual user annotations by replacing the error-prone assignment process by automatic mechanisms. The loop bound specifications generated by the loop analyzer require data structures to be stored. The already existing objective functions and attributes could be reused for this purpose. This results in a shortened development process.

Chapter 6

Experimental Evaluation

6.1 Existing Toolchain

This section provides a brief introduction to the toolchain used in this thesis. It describes the interaction between the individual software modules and gives a simplified overview of the generated outputs. The goal of this thesis is the design and realization of concepts for WCET compiler optimizations. To meet these requirements, the low-level intermediate representation LLIR is enriched with WCET data produced by AbsInt's WCET analyzer aiT.

However, the current toolchain is a temporary solution. The most powerful system is a homogeneous compiler framework that is comprised of a high-level and low-level intermediate representation which directly collaborate with each other. In such a framework, maximal amount of information on the structure and the behavior of the source program is available [BH03]. This data can be passed without any restrictions to the WCET analyzer and enables the generation of most precise timing results. The achievement of this configuration is required for the final stage. For this purpose, the development of a complete compiler framework, which employs the LLIR as backend software, has been started at the Embedded Systems Groups of the Computer Science Department at Dortmund University. However, at the time this thesis was written, the compiler framework was not entirely finished. Thus, a temporary toolchain had to be constructed. Instead of obtaining code from a high-level intermediate representation of the compiler frontend, the `tricore-gcc` is involved. It does not expose all available information stored in its internal representation but its data supply is sufficient for the realization of the collaboration between the LLIR and the WCET analyzer aiT. The existing toolchain is depicted in figure 6.1 and is described in more detail in the remaining section.

The entire developed framework is steered by one central application called `wcet2llir`. It is supplied with a C source code which represents the program to be analyzed. Further input data is the configuration file `wcetrc` that contains specifications of

loops occurring in the source code. After a full run of the framework, a WCET-annotated LLIR file is generated as output.

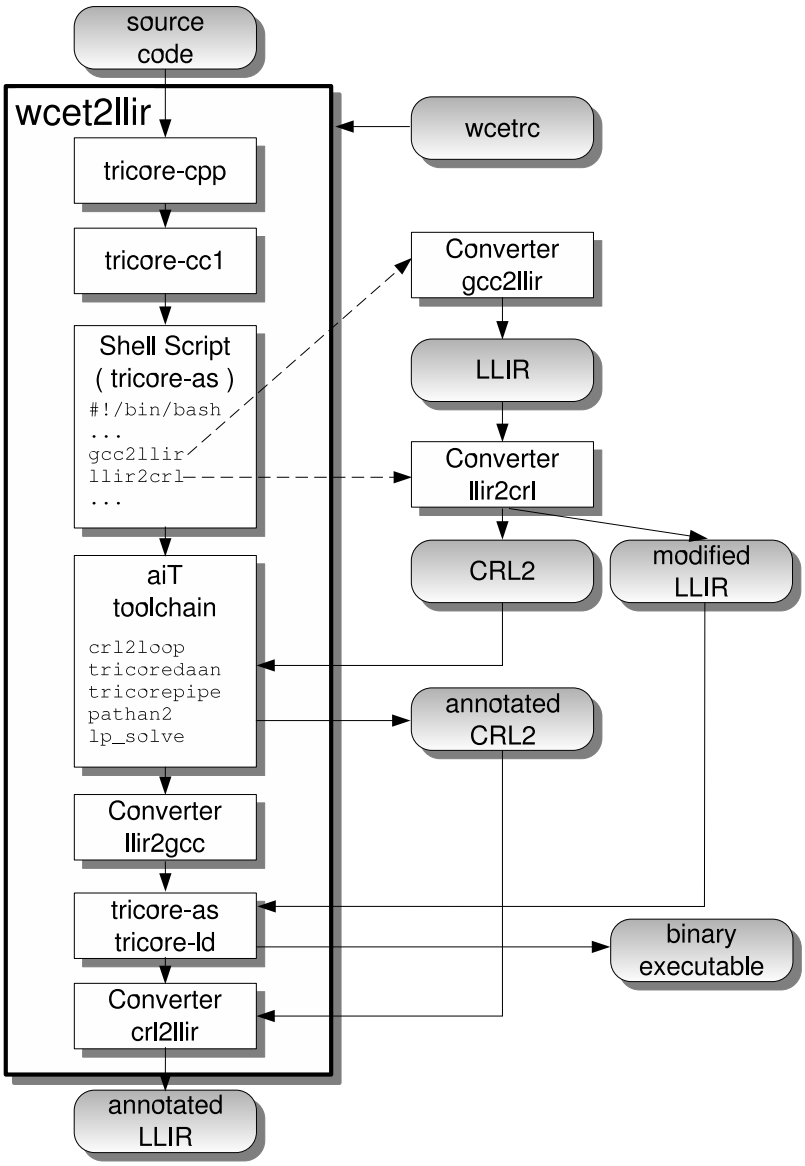


Figure 6.1: Overview of the existing toolchain

In detail, the source code is passed to the `tricore-gcc` compiler. As mentioned above, this approach represents a temporary solution. In the future version of the homogeneous framework, the `tricore-gcc` will be removed and completely substituted by a frontend which cooperates with the compiler backend LLIR. To obtain currently an LLIR representation of the supplied program, the workflow of the `tricore-gcc` has to be modified. The latter is a collection of cooperating applications which serve the purpose to translate source code to machine code. After preprocessing, the TriCore compiler (called `tricore-cc1`) generates assembly code and hands it to the assembler `tricore-as`,

which, together with the linker, translates the code and generates a binary executable. Due to the fact that the LLIR can be constructed from assembly code, the information exchange between the compiler `tricore-cc1` and the assembler `tricore-as` is exploited. More accurately, the original assembler is replaced by a shell script written by [Pyk03] which receives assembly code as input. Additionally, the script invokes the already existing converter `gcc2llir` which reads the assembly input and transforms it to an LLIR control-flow graph. Finally, the converter `llir2crl` is invoked; it reads the created LLIR file and produces an equivalent CRL2 representation. In addition, the converter possibly extends the LLIR by assembler directives (see section 5.2).

In the subsequent step, the central application `wcet2llir` passes the created CRL2 file to the aiT toolchain. It performs a loop transformation (optionally), a loop bound analysis, a value analysis, and a pipeline and path analysis. Finally, the CRL2 is provided to the linear integer solver that generates a WCET-annotated CRL2 representation.

Afterwards, `wcet2llir` invokes the existing converter `llir2gcc`. It transforms the modified LLIR file into an assembly file that is supplied to the assembler and linker. The latter continues the transformations of a regular workflow of the `tricore-gcc` and finally generates a binary executable. This step is optional and may be skipped by providing appropriate command line parameters.

The remaining step is the integration of results produced by aiT into the LLIR. For this purpose, the converter crl2llir is invoked. It reads the annotated CRL2 file and employs the objective-handler framework to interlink the supplementary information with the corresponding LLIR elements. The final result is the sought WCET-annotated LLIR representation. It holds timing information as well as details on existing loops. To obtain an overview of the additional LLIR extensions and their corresponding LLIR elements, `wcet2llir` may be configured to write a log file.

6.2 Results

This section presents the experimental evaluation of the developed software modules. In order to employ the developed framework for realistic applications, its correctness has to be proven. On the one hand, the generated CRL2 representation must be evaluated. It serves as basis for the aiT WCET analyzer. Therefore, it is highly important to supply an error-free CRL2 file in order to guarantee proper analysis results. On the other hand, the integration of the results into the LLIR must be verified. The proper conversion is also highly significant since the imported annotations of the LLIR are involved in further research projects.

Prior to presenting the evaluation results, the workflow of the tests is briefly outlined. In the first phase of the framework, the converter llir2crl analyzes the available LLIR representation of a program and transforms it to an equivalent CRL2 control-flow graph. The

proper operation of the LLIR has been verified in the past and is assumed to be error-free. Therefore, the tests are restricted to the evaluation of the generated CRL2 file. To accomplish this goal, the analyzer results generated on the basis of this framework must be compared with results that are obtained by executing the regular workflow of aiT. More precisely, a program to be analyzed can exist in different representations. The most abstract form, the source code, represents a program at the high-level. After translating it by a compiler backend, the program is given in its hardware-dependent low-level representation. The final step is the compilation of the program into machine code that is presented as a binary executable. All three forms model the same program with equal characteristics.

Consequently, the control-flow graph of the program accommodated in the LLIR resembles the one given in the form of a binary executable. AbsInt's WCET analyzer expects the input program in the CRL2 format which is AbsInt's low-level intermediate representation. This representation may be constructed either by converting the LLIR or by extracting information from a binary. The first option was chosen in this thesis, the second is conducted in a regular run of the aiT toolchain. To sum up, the correctness of the CRL2 file generated by llir2crl is proven by comparing it with the CRL2 file generated by aiT directly from a binary executable. However, an accurate correlation of both versions is not mandatory and even not possible because the executable file provides more information than the LLIR. Rather, a correspondence of the analyzer results based on both CRL2 files is desired. Due to technical reasons, slight deviations of both results are permissable, as will be described later.

In order to compare the CRL2 representations, and thus to prove the correctness of the developed software modules, different types of programs were involved in the evaluation. They are supposed to represent specific classes of source code constructs to cover a large number of real applications. In addition, it was intended to verify the developed converters by applying them to particular applications which are notably found in the domain of embedded systems, e.g. a complex matrix multiplication, an *MPEG4 Motion Estimation* or a *Fast Fourier Transform*. However, at the time these tests were conducted, AbsInt's WCET analyzer for the TriCore processor was still in a development phase. Each extension of processors supported by aiT yields an extensive adoption of the processor-specific WCET analyzer. aiT works in a proper manner for the already fully supported hardware, but allows currently a restricted usage for the TriCore architecture. Staff at AbsInt intensively work on upgrades for the present software and intend a full support for the TriCore processor in the next weeks. In its present state, however, aiT merely enables the analysis of simply structured programs and particularly fails with nested loops. Due to this reason, the analysis of most authentic applications failed and therefore did not enable the inclusion of the embedded system applications mentioned above. Furthermore, it turns out that most of the desired programs written for the evaluation could not be analyzed as well. Therefore, they had to be reorganized and shortened, but still great attention was paid to provide the maximal complexity in order to maximally exploit the analyzer.

Besides the main goal of comparing WCET results, these values emphasize the signifi-

cant impact of the optimization levels applied with the TriCore compiler to generate code. The tremendous reduction of execution time for advanced compiler optimizations arises from two conditions. Firstly, an increase of optimization levels leads to a compaction of the given source code without modifying its semantics. The result is a shorter code with less operations than given in the original source code. Needless to say, this compression yields a shorter execution time of the program. Secondly, aiT benefits from the optimized code. A source code that was compiled with no optimizations may contain structures, notably for loops, which are difficult to analyze. In such a case, it might happen that aiT does not recognize a loop structure. This complicates further analyses and may lead to falsified and overestimated WCET results. In contrast, if standard compiler optimizations are applied, the present code is reorganized by predefined rules which finally produce standard structures. They are recognized more easily by aiT and contribute to the generation of realistic and tight time bounds. Due to this reason, it is recommended to supply code to AbsInt's analyzer that was compiled with at least the first optimization level (-O1).

The conduction of the consistency checks will be described in the following. A program in the form of a C source code is supplied to the developed framework. It generates both a CRL2 file which is supplied to the aiT toolchain as well as a binary executable which was optionally annotated with assembler directives (see section 5.2.5). This binary file is manually supplied to the regular aiT toolchain. In this fashion, analyzer results of both CRL2 representations are obtained. The results from the binary executable are taken as reference and are compared with those resulting from the framework of the present thesis.

It can not be expected that both results are identical. Firstly, the binary executable exposes a wider range of information on the program than the converter llir2crl. For instance, constant global values that are deployed by the value analysis are gained by AbsInt's decoder from the binary executable and supplied to aiT. The LLIR also holds these values, but in its current state there are no direct functions that would return these constants. To retrieve them, a complicated procedure is required. Thus, the constants are omitted in the current version of llir2crl. Secondly, the operations involved in the machine code are not always chosen sensibly by the linker. Therefore, these extracted elements might slightly differ from those that were generated by the converter llir2crl. For clarification, consider the following TriCore assembler operation:

```
ST.W [%a10] 4, %d0
```

To realize this `STORE` operation, the linker may choose between two resembling versions which are provided by the TriCore instruction set:

```
ST.W A[b], off10, D[a]
ST.W A[b], off16, D[a]
```

The specifiers `A` and `D` designate an address and a data register, respectively. `off`*n* denotes an offset value of *n* bits. As can be easily seen, both instruction versions differ solely in the size of their offset value. To refer to the aforementioned TriCore assembler operation, the offset value constitutes 4. Hence, it would make sense to deploy the first version with the

offset value of 10 bits. This decision is made by the converter llir2crl. It assigns operations, which have an offset value smaller than 10 bits, the first instruction version. Otherwise, if the offset value does not exceed 16 bits, the second version is chosen. However, the TriCore linker proceeds differently. It assigns the second version of the `STORE` instruction to the given example. Consequently, AbsInt's decoder, which analyzes the binary executable, extracts exactly the second instruction. As a result, both the CRL2 representation generated by aiT and the one constructed from LLIR data slightly differ. As will be indicated later, this entails minimal deviations in the WCET analyzer results based on both CRL2 files.

The following sections present the results of the experimental evaluation. Due to fact that the set of feasible code is restricted by the current development state of aiT, some simplified test files were taken into account. Each of them will be briefly described and the results are presented in table form. The tests are performed under various conditions. On the one hand, the files were compiled with different TriCore compiler optimizations (`tricore-gcc -O`*n*). On the other hand, the user specifications from the configuration file `wcetrc` were varied. More accurately, the length of the context call string as well as the number of contexts distinguished by the analyzer were assigned different decimal numbers. Increasing values enhance the timing results but, on the other hand, boost the runtime of the WCET analyzer as well.

In addition, to achieve a comparable basis for wcet2llir and aiT, the information which is added by the linker to the executable file has to be omitted. For instance, any calls to the start-up routines like *__main* must not be considered. To prevent such routine calls, the configuration of aiT may be extended by routine names which should not be analyzed. These settings were defined and are applied to all test cases.

Arithmetic operations on array elements

The first test file performs arithmetic operations on integer elements of an array. They are surrounded by a loop. The maximal number of loop iterations as well as the size of the array are defined by the same specifier and the performed operations rely on the current loop counter. The tests were performed with different options. Firstly, the loop bound specifications were modified. The iteration bounds were set to the values `10`, `100` and `1000`. This was achieved by modifying the configuration file `wcetrc` as well as the source code. Consequently, the array size was assigned the same value. Secondly, the configuration specifications `CALL_STRING` and `UNROLL` were adopted. Each test for a particular loop bound specification was performed once with the value pairs (`CALL_STRING=1`, `UNROLL=1`) and in a second run with (`CALL_STRING=inf`, `UNROLL=`*n*) where n corresponds to the current loop bound specifications. The purpose of these settings was to indicate to which extent the consideration of contexts affect the WCET results. The first pair of settings forces AbsInt's analyzer to consider context call strings of a minimal length, namely 1, and to distinguish at most one loop context. These settings lead to highly overestimated results. In contrast, the second run is performed with settings which yield most precise results. In this case, `CALL_STRING` is assigned the value `inf` (meaning no re-

Options	*wcet2llir*	*aiT*
10 iterations, lowest precision [*cyc*]	966	967
10 iterations, highest precision [*cyc*]	957	958
100 iterations, lowest precision [*cyc*]	9516	9517
100 iterations, highest precision [*cyc*]	9417	9418
1000 iterations, lowest precision [*cyc*]	135017	135017
1000 iterations, highest precision [*cyc*]	134018	134018

Table 6.1: Array operations without compiler optimizations

Options	*wcet2llir*	*aiT*
10 iterations, lowest precision [*cyc*]	362	363
10 iterations, highest precision [*cyc*]	360	361
100 iterations, lowest precision [*cyc*]	3242	3243
100 iterations, highest precision [*cyc*]	3240	3241
1000 iterations, lowest precision [*cyc*]	32042	34043
1000 iterations, highest precision [*cyc*]	32040	32041

Table 6.2: Array operations compiled with optimization level *-O1*

strictions to the length of the context call string), and `UNROLL` is set to a value which corresponds to the current loop bound specifications. Of course, equivalent settings were defined in the configuration file of aiT.

Table 6.1 presents results which were obtained by compiling the program without any compiler optimizations. The middle column indicates the results in execution cycles which were calculated by supplying the program to the developed framework called `wcet2llir`. The corresponding results produced by the ordinary workflow of aiT are shown in the third column. Moreover, *lowest precision* is referred to as the settings of `CALL_STRING` and `UNROLL` which are assigned the value 1. In the second case, *highest precision* refers to configuration settings with maximal values as described previously.

The second test series evaluated the same source code which, this time, was compiled by applying the first compiler optimization (`tricore-gcc -O1`). As expected, the original code was highly compacted and leads to tremendously reduced time bounds. Table 6.2 indicates the WCET estimates for the array manipulation.

Further compiler optimization levels produced no remarkable enhancement in terms of the timing results and were therefore omitted. The results from both tables accentuate the proper generation of the CRL2 file by the developed converter llir2crl. The time bounds are almost identical for both aiT and the framework wcet2llir. The minimal deviations arise from the varying generation of operations as described previously. The TriCore assembler translates assembly operations in a less sensible way than the converter llir2crl. It does not exploit the full TriCore instruction set and does not involve the most suitable operation. As can be additionally inferred from the results, the deviating operation is outside the

loop. Despite the increasing loop bound iterations, the results of both frameworks differ in exactly one cycle. Thus, the CRL2 representation which forms the loop is identical in both cases. In spite of these inevitable differences, this issue can be neglected since it does not significantly falsify the analyzer results and still provides a meaningful basis for WCET information.

Moreover, the influence of contexts is depicted. An increase of both the length of context call strings and the number of distinguished loop contexts enhances the precision of the WCET analyzer. The improvements are relatively modest and can be explained by the fact that the test program merely contains one single loop and one function (the `main` function). Thus, the full strength of contexts can not be exploited. As can be seen from the results, the test program, which was compiled at the first compiler optimization level, benefits less from the contexts than the first test series compiled with no compiler optimizations. An interpretation is hindered by the circumstance that aiT is still in the development state and may generate falsified results. One attempt to shed light on the varying results might be the modified program structure which was produced under the affect of the compiler optimizations. The second test series is based on a program code which encompasses simplified loop structures that do not benefit from contexts and therefore do not improve the estimated time bounds remarkably.

Last but not least, table 6.1 indicates a non-consistent sequence of timing results. By increasing the first loop bound specification by a magnitude of 10, the WCET estimates approximately grow by the same factor. However, by multiplying the loop bounds once again by a factor of 10, the WCET results do not increase by 1000 percent as expected but by almost 1400 percent. The reason is not an error in the existing aiT analyses but the absence of a cache analysis and the impact of the code generated by the TriCore compiler. The current version of aiT does not support a cache analysis yet. This feature will be provided in a later version. Due to the missing cache analysis, AbsInt's analyzer assumes that every operation always requires the same number of cycles to be executed, i.e. cache hits and cache misses do not influence the operation execution. Consequently, a proportional increase of the WCET cycles depending on the loop bound specifications would be expected. However, as can be seen from the WCET estimates of table 6.1, the time bounds grow unproportionally. Instead of the expected approximately 95000 WCET cycles for 1000 loop iterations, the aiT analyzer calculated more than 134000 cycles.

This inconsequent result can be explained by considering the assembly code that is generated by the `tricore-gcc`. Comparing the codes of the test file for 10 and 100 loop iterations, the only major difference is a `MOV` operation which handles the loop counter. It operates on the constant value `9` for the first case, and holds the value `99` for the second test case. The remaining code is basically identical. In contrast, the assembly code generated for 1000 loop iterations differs substantially from the previous test cases. Besides the aforementioned `MOV` operation, the basic block, which notably represent the source code loop, highly diverges from the previous test cases. Due to the peculiarities of the TriCore compiler, the source code is translated in a different fashion and the assembly basic

block representing the loop is composed of a greater number of operations. This circumstance clarifies the results given in table 6.1. Because of the lacking cache analysis, aiT treats every operation equally, independent of the actual cache behavior. Consequently, the enlarged code, as a result of the TriCore compilation process, yields increased WCET estimates.

In contrast, table 6.2 shows a proportional increase of time bounds with respect to the growing loop bound specifications. An examination of the corresponding assembly codes generated with the first optimization level indicates a consistent program structure. All three program representations differ essentially in the `MOV` instruction which is responsible for the handling of the loop counter. As mentioned previously, the missing cache analysis leads to a simplified treatment of operations. Thus, the increase of loop bound specifications results in both an enlarged program control-flow graph and adequately increased WCET estimates.

Traversal through multiple loops

This test file is composed of two functions each of which holds two loops. Within these loops, simple arithmetical operations are performed. The `main` function calls the second function and passes a decimal number as argument which serves as loop bound specification. The test order resembles the first evaluation described above. The test file was firstly compiled with no optimizations, secondly the first compiler optimization level was involved. The terms *lowest precision* and *highest precision* denote the combination of the configuration parameters `CALL_STRING` and `UNROLL` (see previous test series). Furthermore, the test with 1000 loop iterations has been performed with a medium precision, i.e. 100 contexts have been considered.

In addition, the loop bound specifications were modified. All tests were performed with global and local settings. A global loop bound specification defines an overall value which is applied to all program loops. On the other hand, local specifications may be used to set individual loop bounds to the existing program loops. For this test series, the global loop bounds were set to `10` and `1000`. The local settings mixed the values, i.e. the loop bound specifications for the `main` function were set to `10` and were assigned the value `1000` for the loops of the second function.

Table 6.3 shows the WCET results obtained when compiling the test program without any compiler optimizations. First, the time bounds verify the proper generation of the CRL2 representation by the converter llir2crl. As can be seen from the table, the time bounds for wcet2llir and aiT are very similar. Depending on the impact of the test parameters, both sets of results change in the same manner. To put it in other words, the influence of the parameters on the aiT time bounds is the same as on the wcet2llir WCET estimates. Second, the results infer that aiT was not able to interpret the present program properly since a decrease of loop bound iterations did not result in reduced WCET estimates. The analysis for the global specification of 1000 loop iterations yielded a time bound of 22025 cycles. After decreasing the iteration number of the first two loops from 1000 to 10 (lo-

Options	*wcet2llir*	*aiT*
global 10 iterations, lowest precision [*cyc*]	245	246
global 10 iterations, highest precision [*cyc*]	245	246
global 1000 iterations, lowest precision [*cyc*]	22025	22026
global 1000 iterations, 100 contexts [*cyc*]	22025	22026
global 1000 iterations, highest precision [*cyc*]	22025	22026
local 10/1000 iterations, lowest precision [*cyc*]	22003	22004
local 10/1000 iterations, highest precision [*cyc*]	22003	22024

Table 6.3: Multiple loops compiled without compiler optimizations

Options	*tricore-tsimb*
global 10 iterations [*cyc*]	3321
global 1000 iterations [*cyc*]	42921
local 10/1000 iterations [*cyc*]	23121

Table 6.4: Simulated runtime of multiple loops compiled without compiler optimizations

cal specifications), consequently a reduction of the WCET was expected. However, aiT produced an implausible time bound of 22003 cycles that resembles the previous test case.

To verify the failure of the aiT analyses and to exclude other disruptive factors, the simulated runtimes of the test program were considered. The goal of this simulation was to indicate that a decreased loop iteration number results in a reduced runtime of the program. Consequently, a decreased program runtime must yield a reduced WCET estimate.

To obtain the runtime of the test program, the TriCore simulator `tricore-tsimb` was consulted. This application is supplied with a binary executable generated by the compiler `tricore-gcc` and produces the simulated number of execution cycles for the input program.

The simulator results are depicted in table 6.4. The tests were performed with 10 and 1000 loop iterations for all loops as well as with 10 iterations for the first two loops and 1000 iterations for the second two loops. The obtained program runtimes indicate the expected program behavior. An increase of the global loop bound specifications from 10 to 1000 iterations entailed a noticeable incline of the total program execution time. Due to the fact that the simulator takes cache behavior into account, an increse of loop iterations by a factor of 100, does consequently not result in a proportionally equal increase of the program runtime. The third test case provides the significant results. It indicates that an intensive reduction of the iteration number for the two loops of the `main` function led nearly to a bisection of the program runtime. Therefore, the conclustion can be drawn that the control-flow graph of this test program was significantly reduced. Consequently, an adequate behavior was expected for the WCET results. However, the results of table 6.3 show that the time bounds were not remarkably affected by the reduction of the loop iterations in the form of local loop bound specifications. This leads to the conclusion that

Options	*wcet2llir*	*aiT*
global 10 iterations, lowest precision [*cyc*]	264	260
global 10 iterations, highest precision [*cyc*]	256	252
global 1000 iterations, lowest precision [*cyc*]	22044	22040
global 1000 iterations, 100 contexts [*cyc*]	22036	22032
global 1000 iterations, highest precision [*cyc*]	22036	22032
local 10/1000 iterations, lowest precision [*cyc*]	10142	10138
local 10/1000 iterations, highest precision [*cyc*]	-	-

Table 6.5: Multiple loops compiled with optimization level *-O1*

Options	*tricore-tsimb*
global 10 iterations [*cyc*]	3042
global 1000 iterations [*cyc*]	16902
local 10/1000 iterations [*cyc*]	10962

Table 6.6: Simulated runtime of multiple loops compiled with compiler optimization level *-O1*

the aiT anaylses failed to interpret the program properly.

Table 6.5 presents the WCET results for the same test program compiled with the first compiler optimization *-O1*. As with the previous test series without compiler optimizations, the similar time bounds for wcet2llir and aiT emphasize the proper generation of the CRL2 file based on the LLIR data. They also indicate that the reduced iteration number for the two loops of the `main` function caused by the local loop bound specifications was correctly recognized by aiT. The reduction of the loop iterations from 1000 to 10 yielded almost a bisection of the time bounds from 22044 to 10142 cycles.

But the results in table 6.5 still confirm the incompatibility of the aiT analyzer. On the one hand, the WCET analysis for the local loop bound specifications, which were performed with parameters to obtain a high precision, failed. On the other hand, there was no reduction of the WCET estimates compared to the test cases that were performed without any compiler optimizations (see table 6.3). Worse yet, the time bounds slightly increased.

It is well known that a program compiled with compiler optimizations yields a decreased program runtime compared to the same program compiler without any compiler optimizations. Therefore, is was expected that table 6.5 presents lower time bounds than table 6.3. However, the comparison of both tables shows that there was no reduction of the WCET estimates for the program compiled with *-O1*.

To verify once again the failure of the aiT analyses, the results of the TriCore simulator are considered. Table 6.6 shows the runtimes of the test program compiled with the first compiler optimization *-O1*. As expected, the reduced program code caused by the compiler optimizations resulted in a decreased program runtime compared to the results in table

Options	*aiT*
global 10 iterations, lowest precision [*sec*]	0.34
global 10 iterations, highest precision [*sec*]	0.40
global 1000 iterations, lowest precision [*sec*]	0.35
global 1000 iterations, 100 contexts [*sec*]	1.56
global 1000 iterations, highest precision [*sec*]	1875.02
local 10/1000 iterations, lowest precision [*sec*]	0.35

Table 6.7: Analysis time of benchmarks compiled with optimization level *-O1*

6.4. A properly working WCET analyzer would consequently produce decreased WCET estimates. However, a comparison of results in table 6.3 and table 6.5 does not foster this assumption. Rather, it violates the logical consequences since aiT generates larger WCET estimates for a program that possesses a reduced program runtime. Hence, it could be shown that AbsInt's WCET analyzer produced falsified results when evaluating program loops.

Furthermore, it turns out that the WCET estimates produced by aiT are lower than those calculated by the TriCore simulator. Comparing the values in table 6.4 and table 6.3 for 1000 loop iterations, it can be seen that the test program takes 42921 cycles to execute in its current state and that its WCET constitutes approximately 22000 cycles. Of course, these values are not feasible since the WCET estimates must always be equal or greater compared to the program runtime calculated by the simulator. The violation the the WCET safeness criterion arises from the current state of aiT. In contrast to the TriCore simulator `tricore-tsimb`, aiT currently does not operate on a realistic model of the TriCore processor core; neither caches, busses and other peripheries are taken into account yet. The enhancement of the processor core as well as the support of the aforementioned hardware features will be provided in the following aiT versions. At the moment, the simplified consideration of the TriCore processor leads to the unrealistic results mentioned above.

Table 6.7 presents the corresponding runtimes of the analyses. The tests were performed using AbsInt's analyzer aiT on the multiple loops program which was compiled with optimization level -O1. The tool provides the output of the total analyses runtime. A system with an AMD Opteron 2.0GHz processor and 2000MB RAM was used as test platform. The time results allow to draw two main conclusions. Firstly, they show that the WCET analysis for a low and a medium precision may be performed in a feasible time. Hence, the analysis can be deployed for real applications. Due to the minimal runtime for even 1000 loop iterations, the WCET can be also estimated for more complex program structures in an acceptable amount of time.

Secondly, the results emphasizes the tremendous increase in runtime when a large number of contexts is taken into account. The test case, which was performed with the objective of achieving the highest precision, took 1875 seconds to estimate the WCET of the program. In contrast, the same program, which was analyzed with the distinction of a single

Options	*wcet2llir*	*aiT*
No compiler optimizations [*cyc*]	185	189
Compiler optimization level 1 [*cyc*]	94	100
Compiler optimization level 2 [*cyc*]	76	81

Table 6.8: Collection of functions compiled with various compiler optimizations

context, required only a fraction of the aforementioned runtime, namely 0.35 seconds. Similarly, the WCET analysis distinguishing 100 contexts yielded a runtime of 1.56 seconds. On the other hand, the difference between the WCET results for all three tests is negligible. Thus, the increased complexity caused by the context distinction is not justified. The enhanced precision of WCET results is out of all proportion to the enormous jump of runtime. Hence, it is reasonable to make a trade-off between the precision and the runtime. As can be inferred from the presented tables, best results are not mandatorily achieved by configuring the analyzer to consider the largest number of contexts.

In summary, it can be said that this test case was still beneficial for the evaluation. Although the produced results are less sensible in terms of the execution time of the program, they indicate that the CRL2 representation created by llir2crl matches the one which is extracted from a binary executable. Moreover, this test indicated the feasibility of the analyses which could be performed in an acceptable amount of time.

Collection of various functions

The last test program represents a collection of functions which perform various arithmetic operations on the program variables. Besides, address operators like C references are deployed. Due to the fact that the program exploits a large number of C operations, it is used to verify the proper generation of various low-level CRL2 operations by the converter llir2crl. Since the program does not contain any loops, this test series is restricted to the variation of compiler optimizations. The latter contribute additionally to the evaluation of a proper CRL2 generation since specific compiler optimizations reorganize the code individually and thus lead to different low-level representations. This, in turn, compels the converter llir2crl to transform a large number of various LLIR operations to CRL2. A correspondence between time bounds of both wcet2llir and aiT emphasizes the correctness of the developed framework.

Table 6.8 presents the analyzer results. All tests were performed with an infinite context call string length in order to guarantee best results. The first row reflects the time bounds for the test program generated with no compiler optimizations. The second and last row indicate the results for the same program compiled with the first and second optimization level, respectively. As expected, more advanced compiler optimizations reduce the code and result in tighter WCET estimates. Moreover, the minimal deviations between the time bounds of the second and third column are derived from the well-known difference of the operation generation between the converter llir2crl and the TriCore linker. They can, however, be neglected since the results still provide significant information on the program

execution time. To sum up, the provided results point out the correctness of the developed converter; despite the slight differences, results produced by both frameworks highly resemble and indicate adequate time bounds.

Back conversion
After supplying the generated CRL2 representation to AbsInt's WCET analyzer, the objective-handler mechanism interlinks the results with the LLIR elements. To prove the correctness of the linking mechanism, the annotated LLIR must be completely traversed and each individual objective handler has to be invoked to retrieve the corresponding objective. Subsequently, the available data accommodated in this information carrier has to be read and compared with the CRL2 file.

For this purpose, the central application wcet2llir was extended by the feature of writing a log file. This option reads the fully annotated LLIR, after it has been processed by the converter crl2llir, and dumps all relevant information into a file. More accurately, it lists all LLIR elements including their supplementary analysis data. In detail, it retrieves the context-dependent execution counts and the belonging WCET estimates including their contexts, the global WCET as well as the loop specifications.

To verify the transformed annotations, the content of the log file was compared with the corresponding CRL2 representation of the array manipulation program used for the first test series. The annotations in both representations matched exactly. Furthermore, the information in the log files and the belonging CRL2 representations of the two other test cases were compared in extracts. Also, they indicated a correspondence. Thus, it can be inferred that the developed objective-handler mechanism operates properly.

It can be concluded that on the one hand, the performed tests successfully showed that the developed converter llir2crl generates a formally correct CRL2 file which enables a proper WCET estimation. On the other hand, the interlinking mechanism converts the CRL2 annotations appropriately to the LLIR. For these reasons, it could be proven that the objective of this thesis was successfully accomplished and that the developed framework wcet2llir may be employed without any restrictions for further research.

Chapter 7

Summary and Conclusions

This thesis presented concepts for the extension of the compiler backend LLIR in terms of the worst-case execution time. For the first time, a successful collaboration between a compiler framework and a WCET analyzer has been achieved. Section 7.1 summarizes the developed solutions and briefly evaluates the results. Finally, section 7.2 concludes the present thesis by discussing ideas for future work.

7.1 Summary and Contribution to Research

Nowadays, embedded systems are ubiquitous in our life. Their fields of application are manifold. In particular, when they serve as real-time systems, which rely on time constraints, the knowledge on the execution time becomes an issue of high importance. Applications that violate predefined time constraints endanger the entire surrounding system. Traditionally, the runtime of a program is determined dynamically by involving measuring devices like oscilloscopes or logic analyzers. However, this approach is error-prone and does not guarantee to find the longest execution time. Static WCET analyses, as the more sophisticated procedure, produce tight and safe estimates and are therefore deemed *future-proof*. For these reasons, it is desired to integrate static WCET analyses into a compiler framework.

To obtain most precise results from the static timing analyses, both high-level and low-level information is required. Due to the fact that compilers process the source code on both levels, they are able to provide all data on the structure and the behavior of a program. Hence, a seamless integration of a WCET analyzer into a compiler framework enables an optimal collaboration. This issue has not been sufficiently solved yet and was the propelling impulse for this work.

The prime objective of this thesis was the design and the realization of concepts for WCET-aware compiler optimizations. To accomplish this goal, this work was focused on

the compiler backend LLIR. More accurately, it was intended to extend the current LLIR by data structures and mechanisms to provide WCET information on the accommodated program. The sought timing data is produced by AbsInt's WCET analyzer called aiT. It is composed of several autonomous applications which analyze the code successively and annotate it with their results.

aiT demands a binary executable program as well as user annotations as input data. The first program of the aiT toolchain, a decoder, converts the object code to a control-flow graph and stores it in AbsInt's intermediate representation CRL2. To establish a collaboration between aiT and the LLIR, this decoding process is skipped. Rather, a conversion of the LLIR data to CRL2 has to be provided. This requirement presents the first main part of this thesis. Both intermediate representations had to be analyzed in order to design and realize a converter that translates a program stored in the LLIR to CRL2. All relevant components as well as the underlying graph structure had to be adequately transformed. The remaining step of the first phase was the appropriate manual invocation of the shortened aiT toolchain.

The result is a unique incorporation of a WCET analyzer into a compiler. The developed framework successfully supports an efficient information exchange by exploiting all compiler analyses data and supplying it to the timing analyzer. To prove correctness, tests have been conducted. They compared the results generated by the WCET analyzer for both program representations, namely the binary executable (regular usage of aiT) and the LLIR. The tests yielded adequate worst-case execution time estimations for both cases and hence it can be inferred that the transformation of the LLIR into CRL2 performed by the developed converter proceeds properly.

To achieve the actual objective of the present thesis, the WCET results generated by aiT had to be shifted back from CRL2 into the LLIR. For this purpose, several concepts were required. The LLIR had to be extended by data structures which accommodate the supplementary WCET information. After evaluating different approaches, it was decided to provide a generic interface between the LLIR and the WCET results. Instead of modifying the LLIR code and assigning the timing results directly to the LLIR elements, a more sophisticated concept was developed. The supplementary WCET data is stored in objectives, which are versatile information carriers, and is interlinked with the backend elements by individual handlers. Unlike the direct modification of the LLIR code, this *objective-handler framework* offers numerous benefits. The objective structure is not restricted to any constraints and may be applied for the storage of various information required by individual applications, e.g. the best-case execution time (BCET) or the energy consumption. Furthermore, due to the separation of the LLIR and the objectives, the latter can be developed independently, i.e. teams are offered the opportunity to work autonomously on the LLIR code as well as on various objective types. Last but not least, the handler, as the interlinking mechanism between the LLIR elements and the gained WCET analyzer results, provides a generic and simplified interface.

The final step to obtain a WCET-annotated LLIR, was the development of a back con-

verter which reads AbsInt's CRL2 representation containing the analyzer results and interlinks the information with the LLIR. Due to modifications to the CRL2 control-flow graph, which were performed by aiT to enhance its analyses, the mapping of LLIR elements and corresponding CRL2 turned out to be a challenge. However, elaborate techniques were developed which enable an appropriate transformation of CRL2 annotations into the LLIR.

The prime analyzer results concerning the execution time of a program are the *execution counts* and the WCET of a single pass trough a basic block. Both values rely on contexts and demand a context-dependent storage in the LLIR. Thus, a new class has been designed that allows the modeling of LLIR contexts. During the transformation of the analyzer results, they are taken into account and serve as additional specifiers. Besides the aiT results concerning the WCET, the analyzer annotates the CRL2 representation with supplementary information which are highly advantageous for further research, notably for the development of a loop analyzer (see section 7.2). Like the aforementioned timing data, details on loops are passed to objectives which are finally interlinked via handlers to the corresponding LLIR elements.

The result of this thesis is a novel WCET-aware compiler backend. After a full run of the developed modules, a program originally supplied in the form of a high-level source code is annotated with WCET information. On the one hand, this timing information can be involved in the development process of WCET optimizing algorithms. On the other hand, it provides a basis for the design and realization of real-time applications. To the best of our knowledge, this work is the first one presenting techniques for a tight and successful integration of a timing analyzer into a compiler infrastructure.

7.2 Future Work

In the current state, the compiler backend LLIR as well as the WCET analyzer are integrated in a framework that employs the tricore-gcc compiler as frontend (see section 6.1). Based on this configuration, the maximal amount of information on the program to be analyzed can not be exploited. To obtain an optimal information exchange, a seamless collaboration between the high-level and low-level intermediate representation is mandatory. Therefore, it is intended to incorporate the developed framework composed of the modified LLIR, the objective-handler mechanisms and AbsInt's WCET analyzer into a homogeneous system.

For this purpose, at the time this thesis was written, the development of a homogeneous compiler framework at the Embedded Systems Groups of the Computer Science Department at Dortmund University was started in parallel. It encompasses a frontend which reads a C source code and hands its representation to the LLIR that finally generates assembly code for the Infineon TriCore processor. Due to the high complexity of developing a compiler, work on the framework has not finished yet. Thus, the tricore-gcc frontend was employed as a temporary solution. However, the current development state of the ho-

mogeneous compiler framework indicates that an integration of the WCET modules will follow in the near future.

The resulting WCET-aware compiler called *wcc* will serve as basis for the development of optimizing algorithms which aim at reducing the worst-case execution time of a program. The emerging timing bounds may be used as indicators for the performance of the algorithms. The final objective is the integration of these optimizations into the ordinary compiler workflow. This would enable a simple and user-friendly usage avoiding tedious user interactions, and possibly the algorithms would evolve into standard techniques for future WCET compilers.

The main drawback of the current framework is the absence of a sophisticated loop analyzer. AbsInt's aiT includes this feature which, however, merely copes with simply structured loops and fails to analyze more complicated loop structures. Hence, the user is often compelled to provide loop bound specifications manually. This error-prone and tedious chore decreases the quality of the software and may even result in erroneous analyzer results. Obviously, it is desired to jettison the user annotations by replacing them by automatic loop bound detections. Thus, the development of an autonomous loop analyzer is a sensible extension. Due to the preparatory work of this thesis (see section 5.5.6), a basis for this intention is already given.

Last but not least, another feature of the aiT toolchain might be exploited. AbsInt's value analysis determines valid value ranges for processor registers at a specific point of the analyzed program. This information resembles the other aiT results and is attached to the CRL2 elements. It could be interlinked with the LLIR by the well-known objective-handler mechanism. This data can be involved in the development of data-flow analysis algorithms. At the current state of the developed framework, the value analysis results are not taken into account. The reason is the absence of appropriate LLIR retrieval mechanisms. In order to enable aiT to generate the sought results, it must be supplied with further program data, other than the information on the control-flow structure, e.g. global program constants. These values are extracted from the assembly file and are stored in the LLIR. However, the current state of the LLIR does not provide any specified functions that would retrieve the sought values directly. The values are stored as LLIR pragmas and require a cumbersome string parsing to be extracted. To obtain these values in a convenient manner, the LLIR needs to be extended by new function. Since these extensions are beyond the scope of this thesis, they have not been taken into account but might be subject of future projects.

Bibliography

[Abs05] ABSINT ANGEWANDTE INFORMATIK GMBH: *Advanced Compiler Technology for Embedded Systems*. http://www.absint.com, December 2005

[ait] *Worst-Case Execution Time Analyzer aiT for TriCore, Manual*. Saarbrücken, Germany,

[ASU86] AHO, Alfred V. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers. Principles, Techniques, and Tools*. Addison Wesley, 1986. – ISBN 0201100886

[BH03] BERNAT, Guillem ; HOLSTI, Niklas: Compiler Support for WCET Analysis: a Wish List. In: *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*. Porto, Portugal, October 2003, S. 65–69

[BW01] BURNS, Alan ; WELLINGS, Andrew J.: *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0201729881

[Byh04] BYHLIN, Susanna: *Evaluation of Static Time Analysis for Volcano Communications Technologies AB*, Mälardalen University, Master's thesis, December 2004

[Bör96] BÖRJESSON, Hans: *Incorporating Worst Case Execution Time in a Commercial C-compiler*, Uppsala University, Undergraduate thesis work, January 1996

[CC77] COUSOT, Patrick ; COUSOT, Radhia: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Conference of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*. Los Angeles, CA, USA, January 1977, S. 238–252

[Cha82] CHAITIN, Gregory J.: Register allocation and spilling via graph coloring (with retrospective). In: *Best of SIGPLAN Conference on Programming Lan-*

guage Design and Implementation (PLDI). Boston, MA, USA, October 1982, S. 66–74

[CRL05] *CRL Version 2, AbsInt Angewandte Informatik GmbH*. http://www.absint.com/artist2/doc/crl2, December 2005

[EEN+99] ENGBLOM, Jakob ; ERMEDAHL, Andreas ; NOLIN, Mikael ; GUSTAFSSON, Jan ; HANSSON, Hans: Towards Industry Strength Worst-Case Execution Time Analysis. In: *Swedish National Real-Time Conference (SNART'99)*. Linköping, Sweden, August 1999

[EES00] ENGBLOM, Jakob ; ERMEDAHL, Andreas ; STAPPERT, Friedhelm: Comparing Different Worst-Case Execution Time Analysis Methods. In: *The Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*. Orlando, FL, USA, November 2000

[EF94] ELSNER, Dean ; FENLASON, Jay: *Using AS - The GNU Assembler*. 1. Boston, MA, USA: Free Software Foundation, Inc., 1994

[Eng98] ENGBLOM, Jakob: *Worst-Case Execution Time Analysis for Optimized Code*, Uppsala University, Master's thesis, January 1998

[Erm03] ERMEDAHL, Andreas: *A Modular Tool Architecture for Worst-Case Execution Time Analysis*, Uppsala University: Acta Universitatis Upsaliensis, PhD thesis, June 2003

[Fer04] FERDINAND, Christian: Worst Case Execution Time Prediction by Static Program Analysis. In: *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 2004

[FV04] FALK, Heiko ; VERMA, Manish: Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization. In: *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2004)*. Amsterdam, The Netherlands, September 2004, S. 137–151

[GCC05] *The GCC Home Page, Free Software Foundation*. http://gcc.gnu.org, October 2005

[GW96] GOODWIN, David W. ; WILKEN, Kent D.: Optimal and near-optimal global register allocations using 0/1 integer programming. In: *Software - Practice & Experience* 26 (1996), August, Nr. 8, S. 929–965. – ISSN 0038–0644

[ICD01] *ICD Low Level Intermediate Representation backend infrastructure (LLIR) Developer Manual*. Dortmund, Germany : Informatik Centrum Dortmund, 2001

[ICD05] *The ICD Homepage – Embedded Systems Profit Center*. http://www.icd.de/es, August 2005

[Inf05] *TriCore 32-Bit single-chip Microcontroller, Architecture Manual V1.3.5*. Munich, Germany : Infineon Technologies AG, 2005

[ISO98] ISO 14882: *Programming languages - C++*. 1. Genevan, Switzerland: International Organization for Standardization (ISO), 1998

[ISO99] ISO 9899: *Programming languages - C*. 1. Genevan, Switzerland: International Organization for Standardization (ISO), 1999

[KA02] KENNEDY, Ken ; ALLEN, John R.: *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2002. – ISBN 1558602860

[KP05] KIRNER, Raimund ; PUSCHNER, Peter: Classification of Code Annotations and Discussion of Compiler Support for Worst-Case Execution Time Analysis. In: *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Palma de Mallorca, Spain, July 2005

[Lan98] LANGENBACH, Marc: *CRL - A Uniform Representation for Control Flow*, University of Saarland, Technical report, 1998

[Lev99] LEVINE, John R.: *Linkers and Loaders*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999. – ISBN 1558604960

[LLPM04] LEE, Sheayun ; LEE, Jaejin ; PARK, Chang Y. ; MIN, Sang L.: A Flexible Tradeoff Between Code Size and WCET Using a Dual Instruction Set Processor. In: *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES'04)*. Amsterdam, The Netherlands, September 2004, S. 244–258

[Mar06] MARWEDEL, Peter: *Embedded System Design*. 2. Dordrecht, The Netherlands : Springer, 2006. – ISBN 0387292373

[MAWF98] MARTIN, Florian ; ALT, Martin ; WILHELM, Reinhard ; FERDINAND, Christian: Analysis of Loops. In: *Proceedings of the 7th International Conference on Compiler Construction (CC'98)*. London, United Kingdom : Springer-Verlag, April 1998. – ISBN 3540643044, S. 80–94

[Mor98] MORGAN, Robert: *Building an Optimizing Compiler*. Newton, MA, USA : Digital Press, 1998. – ISBN 155558179X

[NW04] NIE, Xiaoning ; WAGNER, Jens: *High Performance Network Protocol Processor - Architecture and Tools*. Munich, Germany : Euro DesignCon, October 2004

[Pet05] PETROV, Peter: *Application Specific Processors*. http://www.ece.umd.edu/~ppetrov/ENEE759L_FA05/Intro.ppt, December 2005

[Pyk03] PYKA, Robert: *Retargierbare Bitlevel Optimierungen für den Infineon Tricore Prozessor*, Dortmund University, Master's thesis, August 2003

[Sch86] SCHRIJVER, Alexandeer: *Theory of linear and integer programming*. New York, NY, USA : John Wiley & Sons, Inc., 1986. – ISBN 0471908541

[SW05] STEINKE, Stefan ; WEHMEYER, Lars: *The* encc *Energy aware C Compiler Homepage*. http://ls12-www.cs.uni-dortmund.de/research/encc, December 2005

[The01] THEILING, Henrik: Generating Decision Trees for Decoding Binaries. In: *Proceedings of the 5th ACM SIGPLAN Workshop on Language, Compiler and Tools for Embedded Systems (LCTES'01)*. Snowbird, UT, USA, June 2001

[ZKW+04] ZHAO, Wankang ; KULKARNI, Prasad ; WHALLEY, David B. ; HEALY, Christopher A. ; MUELLER, Frank ; UH, Gang-Ryung: Tuning the WCET of Embedded Applications. In: *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, 2004, S. 472–481

Index

G

H

I

L

M

N

O

P

R

S

T

U

V

W