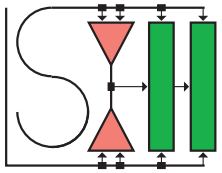
technische universität
dortmund

Diploma Thesis

WCET-centric code allocation for scratchpad memories

Jan Christopher Kleinsorge



Dortmund University of Technology
Faculty of Computer Science XII

September 30, 2008

Advisors:

Dr. Heiko Falk
Prof. Dr. Peter Marwedel

To Jürgen and Monika.

I would like to thank the various people who supported me during the creation of this thesis.

I thank Dr. Heiko Falk for his valuable advice, assistance and motivation.

A special thanks goes to Paul Lokuciejewski and Sascha Plazar for the edutainment and cooperation.

Last but not least, I owe a debt of gratitude to David Fischer, Andreas Schmutzer, Sebastian Senge and Daniel Spitzer for their amicable support.

CONTENTS

CHAPTER 1

INTRODUCTION

The advent of embedded computer systems stems from the ever reducing space requirements while constantly improving the ratio of performance to energy consumption. The widespread use of these systems coined the term *Ubiquitous Computing* [Wei93]. A primary aspect is that embedded devices perform their duty reliably in the background. Often, they are responsible for computing tasks that require *real-time* capabilities. An important field is signal processing. This can be tasks like music, video, radio recording or generation, the control of safety-critical systems like drive controls in automobiles and airplanes, or in medical systems where a precise supervision of patients is required. These tasks can be distinguished by the requirement that a time bound may be exceeded without imposing a ciritical threat to the overall system but causing a loss in quality (*soft real-time*), or the requirement for strict time bounds where a missed deadline can have disastrous effects (*hard real-time*) [Mar03]. A soft real-time system is for example the control of a washing machine whereas triggering the ejection of an airbag timely demands for a hard real-time control.

To deal with hard real-time constraints, it must be possible to analyze the requirements that software imposes to the hardware in terms of computing performance on the one hand. On the other, software that is supposed to be executed on such hardware must be consequently designed to fulfill these demands as well. A key factor are production costs. Software should be able to run on different machines without the need for modifications while maintaining the real-time capabilities of the composed system.

To this end, knowledge of the *worst-case execution time* (WCET) for a given system is crucial. It is an indicator for the worst possible scenario for which design decisions for hardware and software alike can be based on. For optimizations of software with hard real-time constraints, the WCET must be taken into consideration. In general, the optimizations in many existing compilers are only aimed at average case improvements which can be inadequate if it cannot be guaranteed that the WCET bound is adhered. The construction of WCET-aware optimizations is therefore an important field of research.

1.1 Motivation

The memories of many systems form a hierarchy of physical memories accessible through their address spaces. The aim of an optimization can be to make the best possible use of it. Existing optimizations in general attempt to distribute data or code fragements among the given memories so as to achieve improvements. Two paths have been taken until now.

One approach is to utilize the cache memory hierarchy of a system. Caches for data and instructions are components that are very widespread among different architectures. An optimal use of these memories can lead to great enhancements. Also, some architectures offer secondary memories that are not bound to any particular task and that may be arbitrarily used.

Caches have a great advantage over freely usable secondary memories. They posses a hardware control and are tightly integrated with the overall system. That becomes most striking when some form of optimization concerning code is performed. An instruction cache is usually invisible to the processor because it overlaps with another underlying memory in the address space. As no control from the software side is required or possible, no modifications to the program code have to be performed to make use of this faster memory. Fragments of instructions are buffered and overwritten without the need to interact with the cache. Therefore, they are a very efficient means to improve the average execution time of a program.

This simplicity comes at a price. Additional hardware is required to make the cache work. For one, additional memory needs to be reserved to store cache-line tags, and comparison logic needs to be implemented. When it comes to energy reduction, a cache can deteriorate the performance. Moreover, a cache stores fixed sized blocks of memory, which means that also memory objects are stored which are irrelevant from the perspective of optimization. Secondly, there is only a limited number of cache lines and reclaiming lines lies in general beyond the control of the software. Therefore, objects that are important to an optimization can be overwritten by any arbitrary objects that are mapped into the same cache lines. Also, for small programs the cache might not even show a great effect as it will only be filled when an actual access to some memory object has already been performed once. For optimizations related to reducing the worst-case execution time, things get even worse. Since caches are usually fully hardware controlled and come with undocumented or unpredictable replacement strategies, it is often impossible to model a cache hierarchy sufficiently precise so as to be able to predict its behavior. The only way to obtain a safe worst-case time is to assume cache misses on every access. This might lead to a significant overestimation. Techniques have been invented to overcome these problems in some cases. As embedded systems are often required to offer real-time performance and because they are also often required to minimize their energy consumption at the same time, caches are an inherently difficult resource to handle when it comes to the precise modeling of a system to perform any kind of software optimization.

Due to the limitations of caches, some architectures are equipped with fully software-controllable secondary memories. These are memories that are tightly integrated with the CPU to achieve best possible performance. These scratchpad memories (SPM) can be ac-

cessed directly and are therefore in general well suited for optimizations regarding energy consumption and execution time. These memories are directly mapped into the physical address space. When a certain address interval is accessed, the scratchpad is used instead of the main memory.

Modeling certain optimizations is a lot easier with them, as side-effects like the ones described for caches can be neglected now. The strategies to fill this memory can be freely chosen, insofar as there is no unpredictable behavior involved anymore. The lack of control-logic also improves the energy-efficiency.

For WCET-centric optimizations, a scratchpad memory is ideal. The precision of the results of an optimization solely depends on the precision of the model that was employed to reflect the executing hardware. Worst-case time optimizations can rely on predictable timings. To optimize for the reduction of the WCET, assorted parts of a program can be placed into the scratchpad memory.

In this thesis, two different approaches towards the WCET-centric optimization of program code are investigated. On the one hand, the static selection and assignment of program fragments into the scratchpad memory is investigated. On the other, a dynamic allocation is proposed. The aim is to make optimal use of the scarce scratchpad memory space by dynamically transferring selected code fragments between memories at runtime. The compiler framework WCC provides the basic WCET-aware infrastructure, so that these optimizations can be implemented as integrated parts of the compilation process. In this thesis, complete technical solutions to both problems are presented and implemented as integrated optimizations within the WCC.

1.2 Goals

The goal of this thesis is to investigate ways to solve the WCET-directed allocation problem for scratchpad memories. Based on the experiences of similar optimizations for energy reductions using the scratchpad memory and attempts of other authors towards solving this problem specifically taking into account the WCET, new optimization techniques shall be proposed and complete technical infrastructures for their application are to be implemented. Important aspects of the techniques presented in this thesis are:

- *Efficient infrastructure for convenient handling of memory hierarchies.* The presented optimizations all demand for a model of the memory layout of the target machine. This information is typically only available at the linking stage during the compilation process. A framework is to be proposed which enables optimizations on compiler-internal representations with regard to the target memory model.
- *WCET-directed allocation.* It will be investigated why existing allocation schemes - for example for energy reduction - cannot be applied directly. The worst-case time bound needs to be strictly obeyed. This introduces a variety of problems that need to be discussed. A key issue is the dynamic change of worst-case execution paths depending on the optimization decisions.

- *Static scratchpad memory allocation.* This technique attempts to determine a fixed subset of program code which is to be loaded into the scratchpad memory prior to program execution. Challenging aspects of this kind of optimization are the optimal selection of program code and the consideration of technical constraints.
- *Dynamic scratchpad memory allocation.* As opposed to a static allocation scheme, a dynamic allocation approach shall be investigated which allows for the exchange of scratchpad memory contents during program execution. This requires an analysis of the program behavior so that decisions depending on the expected execution paths can be performed.
- *ILP-based modeling.* Both allocation techniques shall be implemented as ILP models. It will be discussed why ILP is a viable way for solving WCET-related kinds of problems like the the enumeration of worst-case execution paths and the modeling of dynamic effects to the optimized program. New ways to model dynamic inflation of the program size and the individual effects to the program performance due to optimization decisions shall be proposed.

1.3 Outline

In the following, the outline of this thesis is presented.

In chapter 2, the formal basics are discussed. The worst-case execution time is formally defined. In addition, it is discussed how the WCET can be determined in theory, and the problems related to this in practice are presented. Also, existing WCET analysis frameworks are briefly introduced.

Chapter 3 deals with the existing tool chain which is utilized at the *Faculty of Computer Science 12* of the *TU Dortmund*. From the analyzers presented in the previous chapter, the aiT WCET analyzer is used as an integral part of the WCC compiler. Both will be presented. The target architecture for the optimizations is the Tricore1 from Infineon Technologies. The relevant aspects of it are introduced as well. It is also discussed which extensions to the existing tool chain are required and what needs to be implemented to make it applicable to the needs of the optimizations proposed in this thesis.

The static allocation optimization will be presented in chapter 4. Related work in the domain of static allocations will be discussed and the requirements of the static allocation will be determined. Among them is the need to obtain a representation of the program structure from low-level program representations. Such a structural analysis is devised. Next, an ILP-based model for the static allocation problem is presented and the technical implementation is discussed.

In chapter 5, an approach towards solving the dynamic allocation problem is proposed. Although parts of the solution can be based on techniques presented in chapter 4, some limitations become apparent. In particular, the need for a global analysis of object lifetimes is motivated and an appropriate algorithm is developed. Afterwards, a solution to the dynamic allocation problem is dicussed which consists of two solution steps. The first is an

ILP-based solution which provides decisions related to memory usage. In a second step, the actual placement of objects within their respective memories is addressed. In addition, technical problems are alluded and possible solutions are discussed.

The results of both optimizations are presented in chapter 6.

Chapter 7 contains a summary of the conducted techniques, and a conclusion is drawn.

CHAPTER 2

INTRODUCTION TO WCET ANALYSIS

This chapter deals with the worst-case execution time (WCET). In section 2.1, it will be formally defined. After that, in section 2.2, different approaches towards the analysis of WCET information are discussed.

2.1 Worst-case execution time

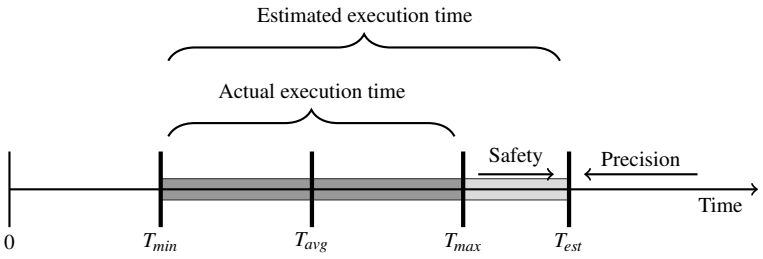


Figure 2.1: The WCET bound

The WCET reflects the maximum execution time of a given program. The value is usually specified in execution cycles. In figure 2.1, its value is reflected by the value of T_{max}. The interval $[T_{min}, T_{max}]$ denotes the total range of execution times that can potentially be encountered. Unfortunately, these bounds cannot be determined precisely. This is due to the fact that the problem of determining when a program terminates is undecidable [Weg99]. Moreso, it is even undecidable if it terminates at all. The best we can do is estimating the the WCET by considering the information available. Running the program multiple times with a variety of inputs only reveals an average time T_{avg}, which is an unsuitable indicator for the execution under real-time constraints. A common approach is to construct an abstract model of the executing machine and to simulate the program's execution. Such an estimation must be **safe** which means that

$$T_{max} \leq T_{est}$$

is guaranteed. A particular problem is that any "abstraction loses information, so the computed WCET bound usually overestimates the exact WCET" [WEE+07]. The quality of such abstractions therefore determines how **precise** the estimations are. Ideally, the result should be as close to the real WCET as possible:

$$T_{est} - T_{max} \rightarrow min$$

2.2 Approaches to WCET analysis

The process of estimating a program's WCET can be broken down into solving two subproblems. Firstly, the actual path through the program which leads to a worst-case behavior is to be determined. Secondly, the actual execution time along this path must be known. Both problems are discussed in section 2.2.1 and 2.2.2 in detail. In 2.2.3, existing analyzers are briefly presented.

2.2.1 Determination of execution paths

The execution path through a program is defined by a sequence *basic blocks*.

Definition 2.2.1. A **basic block** is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them [Muc97].

Given a typical RISC architecture, the first instruction of a basic block can be the entry point of a function, a target of a jump or the first instruction executed right below a conditional jump or a function return. Since they do not impose any side effects of the flow of execution except at its final instructions, a directed graph consisting of basic blocks can be constructed which represents all possible execution paths.

Definition 2.2.2. A **control flow graph** (CFG) is a cyclic, directed graph $G = (V, E)$ whose nodes V corresponds to basic blocks and whose edges E connect two nodes $v_i, v_j \in V$ if and only if v_j is executed immediately after v_i.

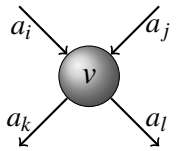
Conventionally, a CFG models the control flow within a single function (*intraprocedural*). Opposed to that, an *interprocedural* CFG models the control flow of an entire program.

Definition 2.2.3. A **program** $PROG = \{P_0, ..., P_n\}$ is the set of all possible paths through the (interprocedural) control flow graph $G = (V, E)$ starting from a unique *source* node $v_S \in V$ and ending in a unique *sink* node $v_T \in V$:

$$\forall P_i \in PROG : P_i = (v_s, \ldots, v_t)$$

To determine the WCET T_{est}, it is essential to know the execution times of the basic blocks as well as the path in the program that maximizes the accumulated costs reflected by the separate execution times. This is referred to as the **worst-case execution path** (WCEP) $P_{WCET} \in PROG$.

One approach towards the determination of this path is to interpret the individual costs as distances between the nodes of the CFG and to solve the *Longest Path Problem* [CLRS01] from source to sink. Unfortunately, solving this problem is NP-complete. Therefore this *explicit path enumeration* can become very expensive quickly. This is due to the typically exponential growth in the number of possible paths.



$$a_i + a_j = a_k + a_l$$

Figure 2.2: Example-constraint for flow-preservation

A widely used technique is the *implicit path enumeration technique* (IPET) [Wil05]. The problem to be solved is to maximize the overall flow through the control flow graph. The variable $a_i \in \mathbb{N}$ corresponds to a node $v_i \in V$ and reflects the number of times a basic block is being executed. In addition $c_i \in \mathbb{N}$ reflects the time it takes to execute this single basic block. The WCET T_{est} of a program is the sum of the individual execution costs of the basic blocks multiplied by their respective execution counts.

$$T_{est} = \sum_{i:v_i \in V} a_i \times c_i$$

The variable a_i needs to be restricted according to the structure of the problem. A basic block can only be executed as often as all of its predecessors together. For every node $v \in V$ in the CFG, the following equation holds true:

$$\sum_{i \in in(v)} a_i = \sum_{i \in out(v)} a_i$$

This equation is also known as the Kirchhoff's Current Law. Figure 2.2 illustrates an example. To determine the WCET, an *integer linear programs* (ILP) [Sch98] can be constructed from these equations. They are presented in detail in [LM95, BR06, WEE+07, EES00].

The IPET is very flexible because besides the base model that constrains the selection of a path, additional constraints can be added easily to model any abitrary supplemental information that could contribute to a more precise result. A drawback is that the WCEP is not explicitly apparent from the result. Also solving an ILP is NP-complete. The complexity of the model as well as the size of the input can have a large impact on the solving times.

2.2.2 Analysis of execution times

Up to his point we have assumed that the execution times of basic blocks are already known. However, this *timing analysis* gives rise to a number of separate problems which shall be discussed now.

One possible timing analysis is to fully simulate the target machine. The fact that the program is actually executed on its full duration makes this *dynamic analysis* a less favorable choice in many respects. The most critical aspect of it, however, is that the execution time can largely depend on the input to the program. So, a safe estimation is not possible in general which draws it generally unsuitable for the determination of the worst-case execution time.

The *static analysis* [Fer04] does explictly not strive for an actual execution of the program. Instead, it is attempted to retrieve a WCET estimation directly from the static representation of a program, like from its binary image file. Usually, different techniques are applied in conjunction to this input to get an estimation as tight as possible. Some of which are:

- *Control flow analysis*: From the static representation of the program a control flow graph is reconstructed. This is fundamental for applying any of the path techniques discussed earlier.
- *Static value analysis*: The input is examined for static information that could indicate what paths are potentially taken during a real execution (*feasible*) or which are guaranteed to be never taken (*infeasible*) [Kir03]. Equally, it is attempted to extract information on iteration counts of loops.
- *Loop bound analysis*: For many loop constructs the determination of exact loop bounds from static data is not possible. In this analysis an abstract interpretation [Cor08] on the relevant parts of the program is performed. This can reveal ranges of iteration counts so that an estimation on the worst-case timing becomes possible because it limits the length of the paths. The worst-case path without bounded loops has an infinite length.
- *Pipeline analysis*: A formal model of the target architecture pipeline is used to estimate its worst-case behavior. Usually, a basic block is considered an atomic unit of execution.
- *Cache analysis*: Similar to the pipeline analysis the possible memory accesses of instructions are tracked and a model of a target architecture's caches is used to obtain estimations on the timinings.

An important aspect of the last two analyses is that in fact the timing is sensitive to the execution contexts. Depending on the state of a pipeline or a cache, an execution of the very same basic block can lead to different timings. Taking this into account helps increasing the precision of the analysis. On the other hand, the consideration of execution contexts significantly increases the complexity of the problem. [Fer04] makes the observation that for loops the execution behavior stabilizes regarding the state of the pipeline or the cache after the first iterations and propose a limit on the instantiation of contexts. Interestingly, this hasn't been an issue in the early days of WCET analyses [PK89] where the architectures were comparably trivial. The estimations were in general even more precise [Wil05].

All these analyses can provide assertions on various aspects of the execution and therefore contribute to the precision of the overall WCET estimation while maintaining its safety.

However, the information which can be extracted from a static representation of a program is often insufficient and an overestimation necessarily results. To cope with this problem two strategies have been suggested.

Typically, the program has been written in an established language. Dealing with a hard real-time constraint is nothing any of the widespread languages, like the popular C programming language, was specifically designed for. [KS86] propose a language that has been specifically aimed at this kind of problem.

Introducing a new language is often no option. To deal with this, some WCET analyzers like aiT [Abs06] can be provided with an external description file that specifically provides information on program properties that can not be determined otherwise. An example of this are accesses to array elements which are typically decided at runtime but are limited to a distinct interval in the address space.

In addition to this, propositions have been made to extend existing languages with so called *flow facts* [EES02] that enable the user to provide this extra information directly in the source code. The compiler then automatically provides that information to an analyzer. A key problem of this approach is that the input can significantly be transformed by optimizations so that the resulting low-level output bears little structural commonality [Eng97]. In [Sch07], the author presents a solution to this problem for an existing compiler framework. Apart from the issues that have been discussed so far, [Lan92] shows that the problem of static analysis is even undecidable in general.

To obtain better results, certain programming techniques are to be avoided, like this is often the case with pointers when used to achieve dynamic behavior [KS86]. Moreso, use of pointers is generally made to alias objects. [Ram94] shows that the *alias problem* alone is undecidable as well.

In general, strict limitations have to be applied to the runtime environment. Any side effects to or by the regular control flow have to be avoided. [LM95] name virtual memory management, interrupts, DMA transfers etc.

Another critical aspect is that many systems are equipped with caches to speed up accesses to instructions and general data. This is similar to the issues just named. A cache implies side effects. During execution, accesses to the main memory are tracked and a logic places chunks of data into a faster but usually much smaller memory. For frequent accesses to the same memory locations this significantly improves the execution performance. Otherwise, if the locality of the accesses isn't high, the positive effect can heavily decrease. Moreso, the cache logic maps the global address space into the limited address space of the cache memory. Different global addresses can possibly be mapped into the same cache locations. The improvements imposed by the cache can therefore heavily depend in the execution history. To the WCET analysis this is critical. The memory accesses have to be modeled as precise as possible. A static analysis can often not determine the access pattern due to runtime dynamic behavior and aliasing. Moreover, the algorithm of the cache logic is not necessarily known either. However, the cache can have a large impact in the overall performance and therefore on the WCET. Depending on the ability to model the cache behavior the precision of the analysis can differ significantly. For this reason some architectures provide software-programmable cache logics or fast on-chip memories which leaves the selection of

contents entirely to the software. The issues concerning WCET and caches are discussed in [CPIM05], [FPT07] and [AP03].

It remains to conclude that the determination of a tight and safe WCET bound is a complex task. The ever increasing complexity of architectures continuously demands for better models. This in turn has an impact on the performance of the analysis. A trade-off between precision and performance seems inevitable.

2.2.3 Analyzers in practice

Numerous research projects are dealing with the problem of WCET analysis. The analyzer HEPTANE [CP01] developed by the IRISA follows a structure-based approach towards the determination of the WCET. The analysis is bound to the syntax of the input program. It is capable to operate on binary or source code inputs likewise and performs its analysis steps in a fully integrated fashion. The timing analysis makes use of pipeline, instruction cache and branch prediction models. However, it is not aware of compiler optimizations which poses a problem to the structure-based analysis. The code is expected to be generated by the GCC compiler. Another framework is *SWEET* [GEE07]. It consists of multiple interacting modules for the different analysis stages. It is capable of determining flow information that otherwise would have to be provided through external annotations. Moreover, it provides different approaches towards the path analysis ("fast path", IPET, clustered). The framework has been integrated into a research compiler. A third project which shall be named here is *calc_wcet_167* [KP04]. Under this project various different tools related to the WCET analysis have been developed. They are integrated into a Matlab/Simulink framework called SETTA. It encompasses analyzers for static analysis and a so-called "measurement-based" technique for dynamic execution time analysis. However, the latter, despite being fast, cannot guarantee safe time bounds. The frameworks are targeted at different architectures and model these with different levels of detail. They can in general be compared by the techniques they incorporate. [WEE+07] provides a thorough comparison of techniques and tools.

There are only few commercial analyzer frameworks available. Examples are *Bound-T* [Tid08] and *aiT* [Abs06] which are both stand-alone WCET analysis tools. They both operate on existing binary program images without the need for explicitly providing a high-level description of them. The former has been developed by *Tidorum Ltd.* and is under contract with the European Space Agency and supports the verification of software for spacecrafts. The analyzer *aiT* has been developed by *AbsInt Angewandte Informatik GmbH* and besides its commercial purpose, it finds its use in research projects at the *Faculty of Computer Science 12* at the *TU Dortmund*. Because of its central relevance to this thesis, chapter 3 dedicates an own section to this tool and its application.

CHAPTER 3

WCET-AWARE TOOL CHAIN

In this chapter the tool chain is presented into which the optimizations presented in the following chapters will be integrated. In the previous chapter, different WCET analyzer frameworks have been presented briefly. The aiT analyzer plays a central role for the determination of the WCET for the optimizations presented in this thesis. It is presented in section 3.1.

Although the aiT analyzer is a stand-alone software, it is used as an integral part of the compilation process of a compiler framework WCC for which the optimizations in this thesis are developed. In section 3.2, the framework itself and the integration of the aiT analyzer are discussed in detail.

WCC is currently aimed at the TriCore1 processor architecture. Basic information of its features and limitations is therefore provided in section 3.3.

In section 3.4, the curret limitations of the WCC regarding the requirements of the optimizations presented in this thesis are addressed. The mandatory extensions to the compiler framework and their implementation presented in detail.

3.1 Introduction to aiT

Among the commercially available WCET analyzer products is the aiT WCET analyzer [Abs06] developed by the AbsInt Angewandte Informatik GmbH. The software is able to perform WCET estimations for a multitude of different architectures and is generally independent of any specific compiler framework. It operates on compiled program binaries and performs a static program analysis.

aiT is a suite of different basic analysis tools and a graphical user interface that allows to conveniently perform overall analyses on programs by automatically invoking and coordinating the actual analysis tools.

From the input to aiT, the control flow is reconstructed by reading the instructions. An internal representation, called control flow representation language *CRL*, is generated. All subsequently invoked tools operate solely on this CRL representation. The employed version of the aiT WCET analyzer operates on CRL2. It is capable to describe items of a program

like routines (functions), basic blocks, instructions and general data. Since a control flow is modelled, each of these components are provided with their respective execution context. The context reflects the current point in the execution history. For example, a basic block within a loop can be executed multiple times. The processor pipeline might potentially be in a different state depending on what iteration it is executed. This differentiation assist in increasing the precision of the WCET analysis. Throughout the steps, the CRL is enriched with the results of the analyses, respectively.

Figure 3.1 illustrates the general workflow of a complete WCET analysis using aiT. We will now briefly investigate which steps are taken to obtain the WCET of a program.

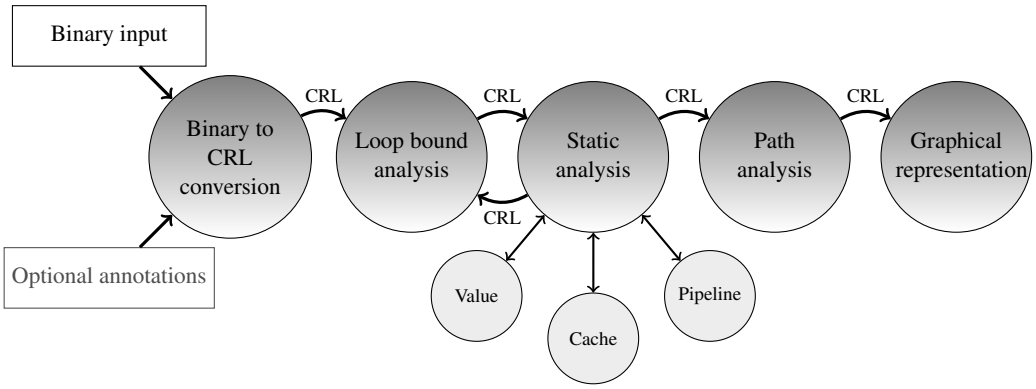


Figure 3.1: Workflow of aiT

The reconstruction of the program's control flow graph is performed during the *CRL conversion*. As can be seen, the CRL serves as the program representation throughout all the analysis steps. Particularly, it is attempted to identify loops that existed in the original source program. This is mandatory to properly model the path of execution through the program, so that the WCET can be correctly estimated. In addition to the binary program, an optional annotation file can be provided. This file contains - for example - specifications for the number of executions of loops (upper and lower loop bound), targets for specific jump instructions, recursion depths and architecture specific information [Abs06]. Annotations for execution flows, such as loop bounds, need to be provided by defining a mapping of the address of a specific instruction to that information. Such a manual annotation is hard to maintain. Figure 3.2 gives an example of such annotations:

```
1 INSTRUCTION 0xa000001e CALLS "Test";
  LOOP 0xd4000400 begin MIN 20 MAX 20;
3 LOOP 0xd400040a begin MIN 20 MAX 20;
  RECURSION  0xd4000448 max 20;
5 INSTRUCTION 0xa00000a8 ACCESSES "Array";
  INSTRUCTION 0xa0000096 ACCESSES "Array";
7 INSTRUCTION 0xd4000470 ACCESSES "Array";
  INSTRUCTION 0xd400042c ACCESSES "Array";
```

Figure 3.2: Example of aiT annotation-file

In the next step, a *loop bound analysis* is performed. Its task is to partly redeem the developer from manually annotating loop bounds by attempting to discover them automatically. In practice, this only works well for very simple loop constructs, so that manual annotations usually remain mandatory.

The loop bound analysis interacts with the *static analysis*. Specifically, the *value analysis* provides possible value ranges for variables. In addition to the loop bound determination, the value ranges serve as input into the *cache analysis*. Since the possible instruction and data accesses are known, an estimation on the performance of one or multiple system caches can be made. To achieve the estimation, a formal cache model is used which can be parameterized in multiple aspects (line width, access times, etc.).

Another integral part of the static analysis is the *pipeline analysis*. A pipeline state is maintained throughout an abstract execution that is performed per basic block. The state of a previous block is taken into consideration when interpreting its successors. Clearly, depending on the current state, the results can be different for one and the same basic block. Because of this, the abstract interpretation of a basic block is bound to a specific context. The number of contexts per block can be configured.

The *path analysis* uses the WCET results from the previous step to calculate the overall WCET of the program. The *IPET* approach, as presented in chapter 2, is used to determine this. In aiT, the ILP is solved with the open-source software *lp_solve* or the commercial *CPLEX*. The overall precision of the result usually largely depends on the proper annotation of the input.

Finally, the results can be investigated graphically. The viewer displays the reconstructed program control flow. The input program can be investigated down to instruction level.

3.2 Introduction to WCC

The WCET-aware C Compiler (WCC) is an ANSI C compiler framework that is specifically aimed at WCET-related development tasks. It has been developed at the *Faculty of Computer Science 12* of the *TU Dortmund*. A key feature is the tight integration of the WCET analysis software aiT. In [EES+99], such an integration is considered the best possible solution for industry-strength development environments. The integration allows to easily obtain WCET information during the compilation process and is the basis for various WCET-related optimizations. The current WCC is aimed at the TriCore1 architecture (see section 3.3).

In the following, the relevant aspects of the WCC compilation process are presented. In section 3.2.1, the stages of compilation are addressed. Between these stages, the input is transformed into two primary representations. A high-level representation called ICD-C IR is presented in section 3.2.2. A code selection stage (section 3.2.3) performs a transformation into a low-level representation, the ICD LIIR, which is dicussed in section 3.2.4. Last but not least, the integration of the aiT analyzer into the WCC framework is presented in section 3.2.5.

3.2.1 Compilation process

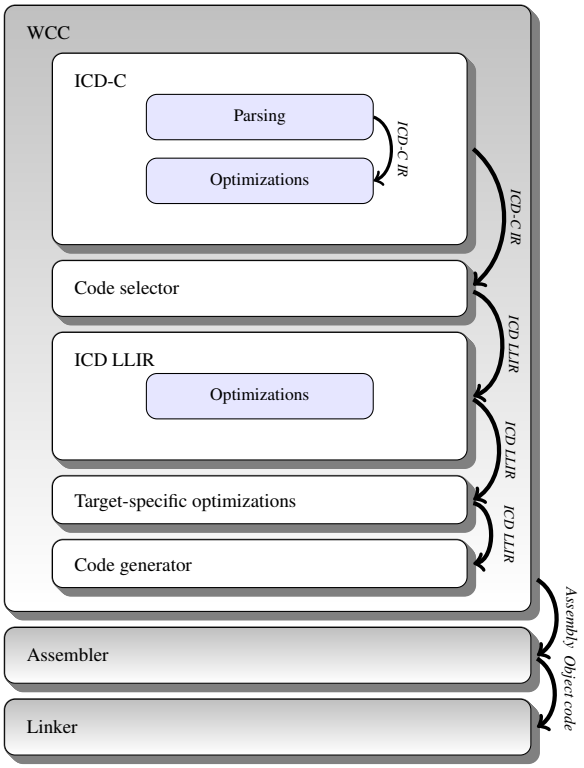


Figure 3.3: WCC compilation stages

As most C compilers, the WCC compilation process consists of multiple stages transforming an input in form of C programming language source files into machine instructions. The various stages in the WCC are illustrated in figure 3.3. During compilation of the source files, two intermediate program representations are generated in succession from the input. These are presented in the following sections.

3.2.2 ICD-C IR

The source files are parsed by a front-end framework called ICD-C which has been developed at *Informatik Centrum Dortmund e.V.* [ICD05]. This stage consists of lexical analysis and parsing of the input. The internal representation generated during parsing is a high-level intermediate representation (IR). Its purpose is to provide data structures that directly represent the input language. That means that statements of the input language (like *if-then-else, do-while*, arithmetic expressions, etc.) have a direct correspondence in form of abstract data types in a connected graph. Figure 3.4 gives an example. As can be seen, the input shown on the left hand side is represented by a directed graph. In this, the separate components identified by the parser are represented by abstract data types.

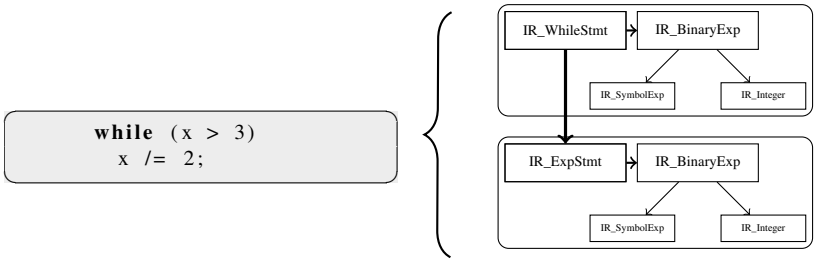


Figure 3.4: Example: ICD-C representation of source-programs

The representation is completely independent of the other parts of the compiler and can be used for arbitrary tasks related to C source code processing. It allows the recovery of the original source program with almost no loss of information. In addition, this IR can be annotated with user-defined information. These *pragmas* are usually stated in the source code to provide additional information to the program. For example, source files to the WCC can be enriched with information that is originally provided as a separate annotation file to aiT, as alluded above. This means that the developer is relieved from providing this information separately. We will investigate the integration of aiT in detail in section 3.2.5. In addition, various optimizations can optionally be applied to this IR which are also part of ICD-C. For the optimizations presented in this thesis the IR is not relevant.

3.2.3 Code selector

The high-level representation is finally transformed into a low-level itermediate representation (LLIR) by the *code selector*. The task of this stage is to optimally match given high-level constructs against patterns of possible instructions that could be generated to provide the necessary semantics in the assembly language. Since often multiple alternatives exist for a transformation from the IR to the LLIR, the best fit regarding the execution performance is selected. The technique is called *tree pattern matching* [FHP92].

3.2.4 ICD LLIR

The low-level intermediate representation (LLIR) generated by the code selection stage is an abstraction of the assembly code that will be generated as the output of the compiler.

Equally to the ICD-C IR, the LLIR can in fact be used independently of the actual compiler. It serves as framework to model any kind of machine instructions. This representation can be used both as an architecture-dependent or -independent program abstraction. It also provides a set of optimizations that are not specifically aimed at a target architecture. As was shown in figure 3.3, the LLIR in the WCC is finally transformed into the actual instructions which are then further processed by an assembler before a linker generates the final executable image. We shall now investigate the LLIR in some detail.

The ICD LLIR is a framework that encompasses the low-level representation and various

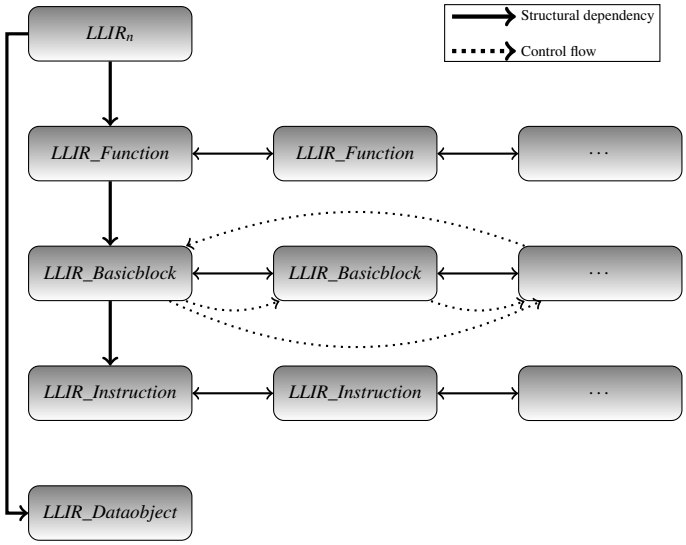


Figure 3.5: ICD LLIR outline

analysis and optimizations techniques. But apart from being a mere container for abstract target instructions, the LLIR provides a hierarchy of types that assist in managing and manipulating the structure of a program.

The most important low-level components of this LLIR are outlined in figure 3.5.

As can be seen, the representation forms a hierarchy of object abstractions. At its highest level, the data type LLIR corresponds to a single compilation unit, which is a container holding all local functions. This is similar to the organization in a C language source program. Functions themeselves are composed of basic blocks. They contain the actual instructions. The ICD LLIR is capable of handling bundled instructions. That is, multiple operations can be part of a single instruction. This serves the need of VLIW architectures. On Tricore1, these operations in fact represent the actual assembly output. Because on this architecture each instruction encompasses just a single operation, it is not explicitly shown in the figure.

Besides forming just linear sequences of functions, basic blocks and instructions, the ICD LLIR also maintains a control flow graph, as it is exemplary illustrated as dotted lines in figure 3.5. This way, the final physical layout of the program can be analyzed by means of linear representations of objects, and also the control flow relations among different objects become apparent.

In addition to this, general data objects have an explicit representation in the ICD LLIR.

The representation is not aimed at a specific architecture. An external machine description provides the necessary information on the actual architecture so that specific instructions with their respective operands can be generated.

As with the ICD-C IR, the LLIR comes with a set of optimizations. These optimizations are built in an architecture-independent way but can be specialized for specific target architectures. To support these optimizations, various analysis techniques have been incorporated.

These are concerned with the detection of relations among objects such as *reachability* and *dominance*, the determination of data live ranges (*data flow analysis*), etc. (see [App97]).

In addition to this, the LLIR allows to attach user-defined information to its objects. As was mentioned in the previous section, also the high-level IR allows this. Thus, it is possible to pass information provided as part of the input source files through the whole compilation process [FL06].

3.2.5 Integration of aiT

The key feature of the WCC is the integration of the aiT software. As was discussed in section 3.1, the aiT WCET analyzer is a suite of analyzing tools that comes with a graphical front-end. This separation allows for a great flexibility. In the case of the WCC, the different stages of the WCET analysis process are triggered as an internal compilation step. The aim of this is to enable WCET-directed optimizations. The intervention of the user to set up a WECT analysis shall be kept to a minimum. The overall goal is to provide WCET information in a convenient way, so that it can be requested at any time during compilation if necessary.

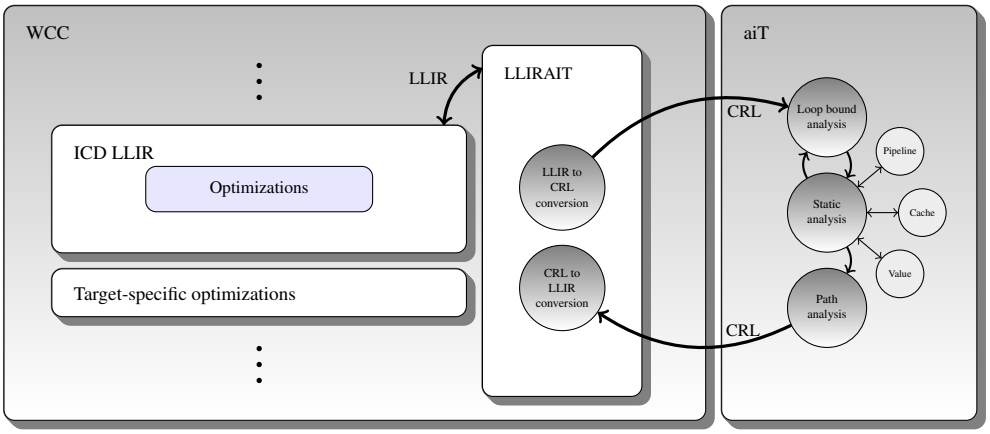


Figure 3.6: Integration of aiT into the WCC

Figure 3.6 outlines how aiT has been integrated. Instead of generating a binary image with a compiler which is read by the WCET analyzer afterwards, the LLIR is directly transformed into the internal format *CRL2* which is used throughout the analysis steps of aiT. After the WCET has been estimated, the results are read back into the LLIR. In general, this process is referred to as *back-annotation*. As was mentioned, the LLIR provides facilities to attach user-defined information to its objects. The CRL does not only contain information on the total WCET after the analysis has completed but, for example, also reflects the WCET for single basic blocks and contains information on the feasibility of control flow paths. All this information is attached to the LLIR objects. Instead of only being able to calculate the overall WCET and to view an annotated control flow graph with the aiT GUI, it is now possible to process this information as part of code optimizations within WCC. The

integration of aiT has been the topic of the diploma thesis of Paul Lokuciejewski [Lok05] and shall not be presented here in all detail.

We discussed how the user of aiT is required to manually annotate loops with their iteration counts or jumps with their respective targets in many cases. This is especially inconvenient for two reasons. Firstly, WCET analysis is usually performed in conjunction with the development of a software that is restricted by certain timing constraints. The fact that aiT must always be run separately to the compilation wastes a lot of development time and the insights gained from the analysis cannot be processed automatically easily. Secondly, any change in the source program potentially results in a different layout of the final binary program. Unfortunately, the user is required to provide the flow annotations with the relocation address of the referenced structure. The user is essentially required to manually look up those addresses and update the annotations.

In WCC, the process of code annotations is fully integrated. Instead of providing a separate annotation file, the user is enabled to provide the required flow annotations as *pragmas* within the C source codes directly. The whole process of manually annotating low-level structures is avoided by passing the annotations through all transformation and optimizations stages of the compilation process down to the LLIR. The integration into the WCC has been developed by Daniel Schulte and was the topic of his diploma thesis [Sch07].

```
2
   _Pragma("loopbound min 5 max 5");
4  for (int i = 0; i < 5; ++i)
     _Pragma("loopbound min 1 max 10");
6    for (int j = 0; j < 10; ++j)
       if (work(i, j))
8        break;
     }
10 }
```

(a) Loop bounds

```
   _Pragma("marker OUTER");
2  for (int i = 0; i < 5; ++i)
     for (int j = i; j < 5; ++j)
4      if (work(i, j))
         break;
6    _Pragma("marker INNER");
     }
8  }
   _Pragma("flowrestriction
10           1*INNER >= 15*OUTER");
```

(b) Flow restrictions

Figure 3.7: Flow facts in C sources

Flow facts in WCC basically come in two flavors. Figure 3.7a shows a snippet of C code that has been annotated by so called *loop bounds*. An annotation is always provided with a *pragma* statement directly in front of the statement to be described. In the example, the two loops are annotated with their minimal and maximal loop iteration counts, respectively, A more elaborate approach to the specification of execution counts is shown in 3.7b. Instead of direct annotations of statements, *markers* can be freely placed in the source code. In addition, so called *flow restrictions* define the relation of execution counts of markers in the control flow of the program. They can be used to model arbitrarly complex scenarios where iteration counts are not obvious. Examples are complex loop nests or recursions.

In addition to manually annotating the source files, a loop analyzer can be used to automatically detect the iteration counts of loop constructs. Although aiT itself comes with a simple loop analyzer, Daniel Cordes developed an analyzer specifically for ICD-C for his diploma

thesis [Cor08].

The WCET analysis is available as a library within the WCC. It does not only serve to determine WCET information at a specific point during the compilation process but can be used abitrarily within the optimization stages.

3.3 TriCore1 architecture

The basics of the optimizations presented in this thesis are largely independent of a specific target architecture. However, the technical solutions presented later are all aimed at the TriCore1 architecture. We will now summarize the most important facts about it.

The TriCore1 is a RISC processor aimed at embedded systems developed by *Infineon Technologies*. It features DSP as well as general-purpose capabilities. Like most RISC architectures, it uses a load/store scheme for memory accesses. The instruction set architecture (ISA) is based on 32bit and 16bit wide instructions. In many cases, one and the same operation can be performed with a 32bit wide instruction or a 16bit wide variant. The latter is limited in terms of operator sizes or utilizes predefined registers implicitly. The smaller instructions significantly reduce the consumption of space and energy. Apart from a set of general purpose instructions, it features a multitude of DSP-related ones. This includes saturated arithmetic as well as extended abilities to perform bitwise operations.

This processor features two major and one minor pipeline. The former are both four-stage pipelines, sharing the fetch stage. One of them is dedicated to integer arithmetic only, the other to loading and storing. In theory, all instructions but ones related to multiplication and branching can be executed within just one cycle. The minor pipeline serves the execution of zero-overhead loops. This allows to largely avoid multicycle instructions. Typical problems at this point stem from data dependencies and accesses to slower memories. Both major pipelines will stall if only one of them has to wait.

There exist 32 general purpose registers (GPR), divided into 16 data and sixteen address registers, each of which is 32bits wide. A predefined mapping allows to access a pair of GPR by means of a single extended register which makes continuous, 64bit wide registers generally available.

Its address space is limted to 4GB, equally divided into 16 segments. It can be accessed by means of physical addresses or by optionally utilizing the virtual memory capabilities. The TriCore1 is a full-featured multi-tasking CPU. Therefore, it provides common capabilities to define and restrict per-process resources. The virtual memory is part of these mechanisms.

Branching on the TriCore1 is supported by means of a static branch prediction. Figure 3.8 illustrates the predicition scheme. As can be seen, all 16bit wide jump instructions are considered *taken*, as well as 32bit wide jumps with a backward displacement. These two cases cover situations that usually occur when it comes to skipping short sequences of instructions[1] or for repeating loop bodies. As opposed to that, a 32bit wide instruction

[1] This frequently occurs in code generated from the C language for many conditional statements.

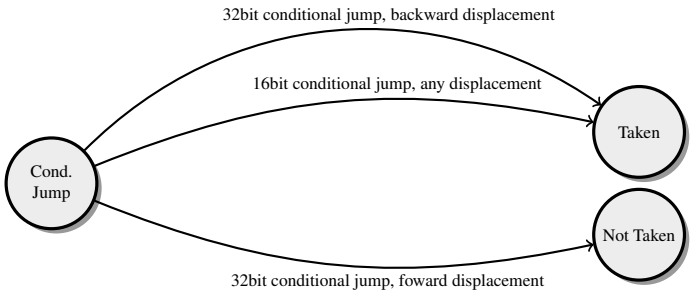


Figure 3.8: Static branch predicition

performing a forward displacement is considered *not taken*. Depending on the prediction and the actually issued instruction, a jump can cause a delay in the execution.

		Predicted	
		Taken	**Not Taken**
Issued	**Taken**	1	2
	Not Taken	2	0

Table 3.1: Static branch prediction and timing

Table 3.1 lists the delay cycles resulting from the static branch prediction depending on whether the prediction held true. As can be seen, a stall of up to two cycles can potentially occur. Mispredictions can significantly degrade the overall performance. In particular, since a stall in one of the major pipelines causes the other to stall as well.

The specific implementation of the TriCore1 that is supported by the WCC is the TC1796. Figure 3.9 illustrates the primary memories and busses. As can be seen, separate physical memories are dedicated to instruction codes and data. These memories are all on-chip.

Instruction codes are executed right out of the *flash memory*. This access can be cached utilizing the *instruction cache*. The *scratchpad memory* is tightly coupled to the CPU core and therefore allows 1-cycle access permanently. As opposed to instructions, the *flash memory* dedicated to data is not cacheble at all. In addition two memories, the *local memory* and the *scratchpad memory* exist for fast accesses. The latter allows for 1-cycles accesses. Also, a separate *stand-by memory* exists to keep data in low-power states.

For the optimizations that are described in this thesis, only the *flash memory* and the *scratchpad memory* for instruction codes are relevant.

3.4 WCC extensions

In the upcoming sections, extensions to the WCC are presented. In the light of the requirements of the optimizations presented in this thesis, certain limitations in the exising implementation become apparent. Specifically, modeling of the target memory hierarchy

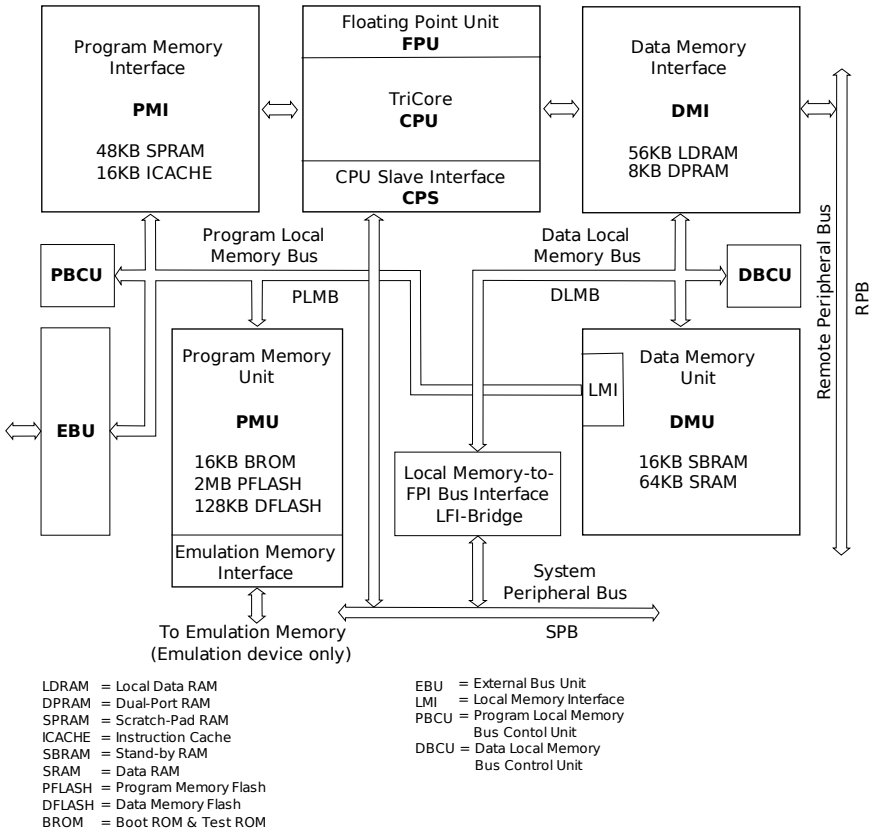


Figure 3.9: TriCore1 architecture

requires a rework. In section 3.4.1, the current implementation is addressed and it is motivated why an extension is required. The additions are then dicussed in detail in section 3.4.2.

3.4.1 Existing framework

In section 3.2.5, we discussed the integration of the aiT WCET analyzer into the WCC. It was explained how a user can provide flow fact annotations within the source code which are then provided as input into aiT. It was noted that such annotations originally had to be provided by means of an external file that mapped the annotations to specific memory addresses. In the original implementation of the aiT integration, these adresses are provided as object attributes into the CRL. Due to the structural similarity of the LLIR and the CRL representations[2], addresses can be assigned directly. Specifically in the CRL, objects like functions, basic blocks or instructions posses an address attribute which is to be specified. The addresses are calculated by accumulating the sizes of the LLIR objects, starting from the base address of a specific memory. For example, the default setup uses the base address

[2]Refer to [Lok05] for details.

of the program flash memory which is illustrated in figure 3.9. These addresses need not be logically correct but need to be valid. For code objects they are valid only if they point to an address range which is readable, executable and do not overlap. Data objects in contrast may overlap. Therefore, the address attributes for code objects are obtained by summing up instruction sizes. For data, even only a single address is provided for all objects.

This approach was sufficient for simple analysis tasks but lacks flexibility in a major aspect. The analysis of instruction codes that execute from different memories is impossible since all corresponding addresses are set off from a fixed base address of a single memory.

This is a problem we have to overcome since the optimizations we will propose in the following chapters of this thesis specifically have to take multiple memories into account. In this section, an extension to the aiT integration is presented which allows to model and analyze arbitrarly distributed program objects within a freely definable memory hierarchy. This work also contributes to solving the problem of indeterminate data accesses although this is not of a concern in this thesis. Felix Rotthowe extended the LLIR in this regard. This is thoroughly discussed in his diploma thesis [Rot08].

The requirements for such an extension are that a flexible yet simple memory model can be defined and that objects from the LLIR can conveniently be assigned to those memories to the effect that the aiT WCET analysis can rely on this information.

The fundamental idea of this extension is to make information on the memory model available in the optimization stages of the LLIR. At its current state, the LLIR enfolds machine information only to the extent of CPU related information like instructions, their operand encoding and registers. The output of the LLIR is assembly code which is translated into object code by an assembler (figure 3.10). This binary output is still independent of the actual memory layout. Only in a final step, a linker relocates the object code according to a memory description to obtain an executable binary.

To enable a user to define such memory assignments during a compilation step, the information that is normally only available to the linker in a usual compilation process needs to be provided already within the compiler itself. This is motivated by the fact that the integrated WCET analysis requires detailed information on these assignments. The stand-alone implementation of aiT used a readily relocated binary which allowed it to obtain the required address right from the image. Now that aiT is used as an integral part of the compilation framework, the relocation steps being omitted need to be simulated to achieve the same degree of flexibility.

At this point, we shall present the terminology used in the upcoming discussion. The implementation of the address handling and relocation on the LLIR is guided by the workflow found on the GNU platform [Fou08]. The formats and conventions comply with the EABI specification for the TriCore1 architecture as defined in [Inf07]. The assembly instruction codes emitted by the WCC compiler directly correspond to the instructions defined in the LLIR. In addition, assembler directives are generated that help organizing such code concerning their assignment to specific memory addresses. Such directives define *sections* that serve as abstract containers. The TriCore1 EABI defines a predefined set of such section names for instructions (*.text*) and initialized data (*.data*) in general, for read-only data (*.ro-*

data) and uninitialized data (*.bss*[3]). When the code is assembled, the binary codes will be assigned to their respective sections in an *object file* as defined by the ELF specification [TIS95]. This is shown in figure 3.10.

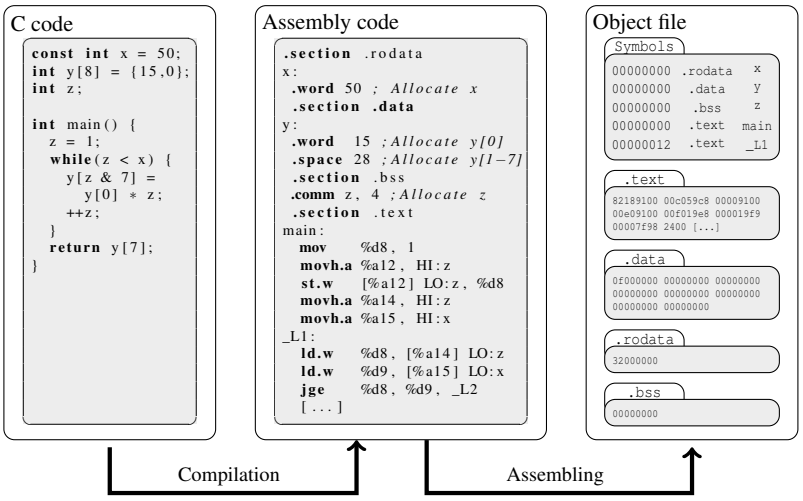


Figure 3.10: Object file layout from assembly code

From every compilation unit, a single object file is generated. For all references between objects that require absolute addressing, an entry in a separate table is created. Absolute addressing is always required for references across sections, including references to objects not even in the current compilation unit. As opposed to that, all relative references can already be resolved. These occur for the majority of jump instructions on TriCore1.

Every such object file is equipped with at least the binary representation of the assembly code organized in sections and a *symbol table*. The latter maps object names to addresses relative to their respective sections and also contains entries for unresolved references.

To generate the final program binary, all object files that make up the program are combined by the linker. The WCC utilizes the linker *ld* from the *GNU binutils* [Pro07] package. The following description explicitly refers to its features.

To model how this combining is to be performed, an external description file (referred to as a *linker script*) is provided. Within such a file, so called *SECTION* directives define how the source sections shall be laid out in a single target section. An example is given in figure 3.11. Here, all sections with the prefix ".text" are combined into a single section labeled ".text", each additional section is placed on top of the previous one in no specific order, aligned to a 1024 byte boundary and with the gaps that could occur within this address space filled with a bit pattern of "0xff". This shall only serve as an example.

Next, the combined sections can be assigned to a specific memory address region. The same linker script in general also contains a description of the address space layout by means of *MEMORY* directives. With these, address space intervals can be specified. It is defined by

[3]Historic naming. Originally "Block Started by Symbol", but has no such meaning in this context.

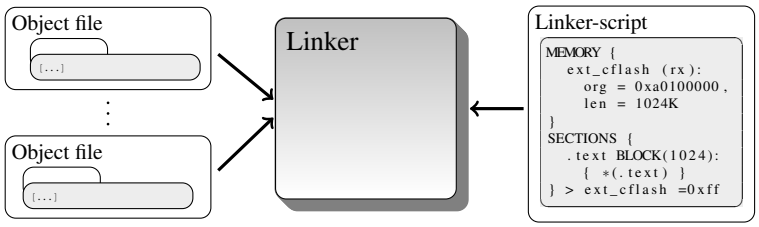


Figure 3.11: Example of *ld* linking process

its base address and a length. As can be seen in the example, the section ".text" is assigned to the memory named "ext_cflash". Summing this up, the linker script describes how a program should be loaded into the memory and takes actions accordingly.

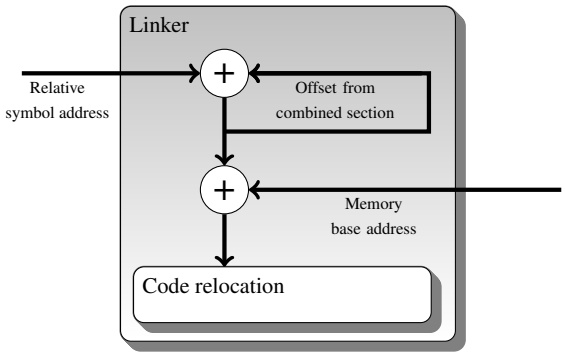


Figure 3.12: Example of address translation

While sections are combined, unresolved symbols from one object file are matched against the symbol tables of others to obtain the actual addresses. These depend on the placement of source sections into the target section and are relative to the latter. A new symbol table reflects these changes. The calculation of the final addresses is drafted in figure 3.12. Apart from the simple accumulation of object sizes, a multitude of attributes can take effect, which is not shown in the figure.

Due to the assignment to memory regions, it is now possible to calculate the final, absolute addresses that will be used in the executable program image by setting off all section-relative addresses by the target memory's base address. The affected address references within the instruction code are patched accordingly. Now that the physical addresses are set, the symbol tables can be discarded. All symbolic references should be resolved at this point and the final binary image can be generated. [4]

[4]Dynamic linking techniques are irrelevant at this point.

3.4.2 Framework additions

What has just been described is the typical workflow that occurs for stand-alone TriCore1 programs. Such an image can afterwards serve as the input to the aiT WCET analyzer. Obviously, it is advantageous that all address references within the instruction code are now fully known. This way, it is easy to determine where memory accesses take place. The problem is that it is unknown what objects have been accessed, which is the reason an external annotation file was needed to describe this. On the other hand, due to the integration of aiT, information on objects being accessed can be conveniently provided instead but the physical layout is unknown. Because of this, we will extend the LLIR to offer the ability to dynamically model the section assignments and the memory layout, and to provide symbol tables to enable a user to conveniently access address information on most objects abstracted by the LLIR. The role of the symbol tables in this particular case is to deliver information on absolute, memory layout-dependent addresses to aiT so that it operates exactly as if a readily linked binary image had been provided as its input. This extension restores its full capabilities.

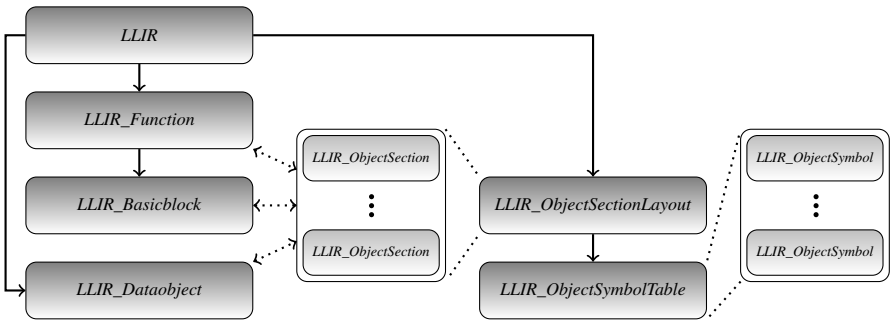


Figure 3.13: ICD LLIR extensions to model object file sections.

As was illustrated in figure 3.5, the top element of the low-level hierarchy is a type named LLIR. It corresponds to a single compilation unit. In the design of the extensions, the workflow, as described in the previous section, is adopted. Many attributes that can be specified in a linker script can also be applied to the ICD LLIR. Each abstract data type of the LLIR is responsible for managing the assignment to sections for its subordinate types. This is true for LLIR functions, basic blocks and data objects. Although also instructions could be easily assigned separately, it is hard to imagine a use case for this and it is entirely unnecessary for the specific needs of this thesis. The extensions for the LLIR itself are shown in figure 3.13. A type called *LLIR_ObjectSectionLayout* is attached to the LLIR. It is responsible for managing the separate objects of type *LLIR_ObjectSection* representing single sections. Sections can be dynamically created or destroyed at runtime and hold attributes that exactly match the ones that can be specified in linker script SECTION directives. The LLIR types representing functions, basic blocks and data objects are extended to hold references to section instances. They can be dynamically attached and detached from them. Before these changes, the code generator was responsible for generating assembly directives. These were static and conformed to the EABI specification. Now that the LLIR is entirely responsible for this, a set of default section objects is generated and the affected LLIR objects are at-

tached to them upon initialization. An important requirement of the extensions was that the whole section management remains fully transparent to guarantee full backward compatibility. Software that made use of the unmodified ICD LLIR has to be kept in fully working condition without any changes.

In addition, a symbol table named *LLIR_ObjectSymbolTable* was introduced. Its basic purpose is to provide a mapping from symbols to memory addresses. Such symbols can be function names, labels of basic blocks, names of variables but also ones that have no direct correspondance in the LLIR type hierarchy like markers for certain places in the address space. Per LLIR, there exists only a single symbol table which lists all of its symbols. The table is sensitive to sections and distinguishes LLIR-local symbols like basic block labels from external symbols like function names[5]. The symbol table also allows for reversed lookups, which means that names can be looked up by providing a section name or an address.

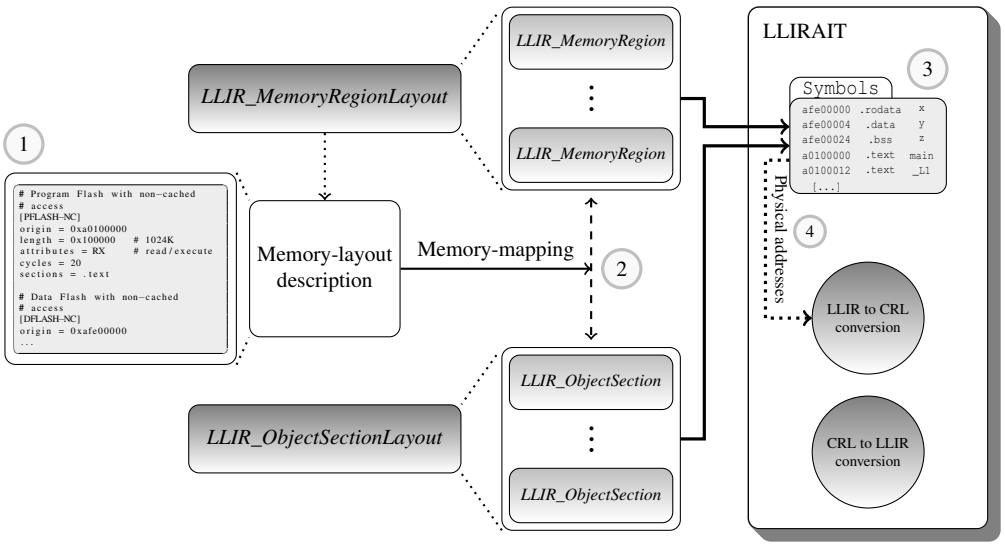


Figure 3.14: Extension to LLIRAIT

At this point, all addresses are calculated relative to the currently assigned section. To calculate the final addresses, the memory layout must be taken into account. Similarly to the linker script, this can be modeled by an external layout file. Figure 3.14(1) illustrates how the format looks like. Similarly to the design of the section layout, a type called *LLIR_MemoryRegionLayout* is responsible for the dynamic management of a set of *LLIR_MemoryRegion* types. The layout manager reads the external description and generates the region objects accordingly. Again, the possible attributes conform to the ones that can be specified in a linker script.

The section layout and the region layout in combination are generally capable to abstract from a fully featured linker script and it would be entirely possible to generate one from just

[5]Excluding C functions attributed *static*, therefore also being only locally visible.

the information now available through these extensions. This results in a large flexibility concerning optimizations that are specifically aimed at utilizing memory hierarchies, such as the ones that will be presented later in this thesis.

The final issue to be solved is how the individual sections can be assigned to their respective memory regions, so that the calculation of absolute addresses becomes possible. The description file for the memory layout also includes a mapping attribute that defines which sections shall be included in the regions (figure 3.14(2)). This is different from the workflow in a linker script where the sections per object file are usually first assembled in a single section and then assigned to a memory region. This mapping determines the assignment of all sections with a specific name from all LLIR instances into a single region. Equally to the actions taken by the linker, the objects are assigned to a region sequentially taking into account attributes like the address alignment. After this, the final addresses can be calculated (figure 3.14(3)). The individual addresses of objects relative to the start of their section, the offset imposed by the assembling of sections and the base address of the target memory region are taken into account to fill a symbol table which serves as input to the LLIR2CRL-conversion directly (figure 3.14(4)).

In conclusion, we have now fully modeled the flow of the information that is relevant for an aiT WCET analysis from an internal representation to the point where an executable binary file would be available. The library that manages the translation from the LLIR into the CRL is adopted to this new infrastructure. The static memory model that was originally applied is discarded.

CHAPTER 4

STATIC SCRATCHPAD ALLOCATION

In this chapter, a program optimization technique is presented that aims for a reduction of the WCET by utilizing a scratchpad memory (SPM). The general idea is to allocate selected parts of the instruction code to that memory to achieve improvements.

In section 4.1, we will first investigate techniques that have been applied to SPM-related optimizations in general. We will come to the conclusion that modeling an ILP is a a viable and well understood method in this regard. In addition, generally WCET-directed optimizations will be discussed briefly. From this, we can conclude that an ILP approach with the goal to solve our problem along implicit paths through the program is promising.

Next, in section 4.2, a technique called *structural analysis* is presented which will assist in the determination of the program structure. The need for such an analysis is motivated. It will also help in understanding problems particularly related to the low-level representation of programs within the WCC. Ways are shown to overcome these.

Subsequently, the optimization problem is formally described (section 4.3) and an ILP-based model is motivated that will serve as the basis for the actual optimizng model which is presented afterwards in section 4.4. There, a complete ILP model is formalized that will address all the specific problems we previously discovered to solve the static allocation problem.

Afterwards, in section 4.5, the workflow to establish the ILP as an integrated compiler optimization is discussed. It will be shown how the analysis presented in 4.2 is utilized to generate the model and what additional steps are required to apply the results from the ILP model to the input program.

4.1 Related work

In the domain of static allocations of objects to different memories various different approaches have been pursued. The basic idea has always been to enhance a system's performance by smartly utilizing the existing memories. Performance is of course not restricted to the decrease of the WCET. Most approaches are concerned with the enhancement of power, time or space requirements. According to [AMF+04], different ways have been taken to achieve the aforementioned effects. All ground on the idea to utilize different kinds of

memories to achieve improvements. However, specifically for optimizing program code, a certain commonality can be seen. The basic idea is to select basic blocks, sequences of basic blocks or complete functions under certain criteria and to place them into a faster memory so an improvement crops up. For optimizations related to the reduction of energy dissipation, the most expensive basic blocks are chosen. As pointed out in [VWM04a] and [SWLM02], simply picking a greedy strategy can lead to suboptimal or even worse results. In addition, the overhead created by placing the code in a different memory has to be considered. An improvement only occurs when the placement of the program code into a different memory exceeds the costs of the overhead. To solve this problem optimally, an ILP is used. It allows to state the problem as linear equations that constrain a set of integer variables while trying to maximize or minimize an objective function that depends on these variables. Among all techniques proposed for these kinds of optimization problems, ILP is the most popular one. This is because it is easy to model costs expressed as execution cycles or energy consumption. The aim is always to either maximize a gain or to minimize the costs. Additional costs related to certain assignments of values to variables can easily be intergrated by simply adding new equations that further constrain the problem.

One approach to optimization is to statically lock contents in the cache. Cache lines are explicitly filled, then the cache is locked. A requirement is that the cache can be software-controlled in the desired way. When this is possible, the cache can be treated almost like a general purpose memory, except that the cache logic needs to be taken into account. This method requires that the system allows for controlling the cache in such a way. Therefore, this method is not generally applicable. Given that a cache locking optimization is to be performed, the cache itself has to be modeled as an ILP. Using a scratchpad memory significantly reduces the complexity of the problem. However, they are not as widespread as caches.

Concerning WCET-centric optimization, several techniques have been proposed concerning cache locking. A particular problem with WCET optimizations is that simply picking the most expensive basic blocks (regarding their execution time) is insufficient because they may not lie on the most expensive execution path at all. This issue will be discussed in depth later. In [CPIM05], a genetic algorithm is presented that aims at optimally solving the assignment problem. However, this does not necessarily yield optimal results. Another approach has been presented in [FPT07]. An explicit search for the most expensive paths is performed and an optimization along this path is performed. It relies on repeatedly investigating the control flow graph is therefore expensive to perform. Another interesting approach has been presented in [Pua06]. Here, optimizations along the most expensive path is performed multiple times. After a certain number of steps, the partially optimized program is analyzed again, so that all information on the costs of basic blocks is now updated. The most expensive path is explicitly recalculated and the optimizations continue.

To overcome the problem of changing worst-case paths, [SMRC05] propose a simple ILP scheme that is able to implicitly model such paths. This approach is well suited for this kind of problem as explicit path searches can be very time consuming. Reducing the number of reevaluations, however, can lead to a loss of precision. The presented approach is limted to data allocations. Still, the fundamental approach is interesting and we are getting back to it further below.

4.2 Program analysis

In this section, a technique will be presented that fundamentally supports our efforts to optimize instruction codes of a program. Prior to performing any modifications on the input, a thorough knowledge of its structure is vital. Independent on the actual optimization we will present later, the discussion in section 4.2.1 will emphasize the problems encountered for inputs as low-level program representations and motivate the need for an extended analysis. Briefly, two well known alogrithms for program analysis will be presented. The deficiencies observed will motivate the use of a different approach than these. Namely the *structural analysis*. This technique will be presented in all detail in section 4.2.2. Since this gives us an intimate understanding of the structure of the input, it is presented prior to the actual optimization.

4.2.1 Overview

Performinig program optimizations related to the control flow of a program requires substantial information on the structure of the code.

This structural information can encompass the control flow between pieces of code within a single function as well as information on how functions are related to each other. Moreover, in code generated from a structured programming language like C, an understanding of how structures are nested is required so as to be able to identify, manipulate or replace code that is included in those.

The ICD LLIR does not carry any information on what high-level constructs transformed into which instruction fragments. Therefore, it is in general not possible to easily reconstruct a high-level view of the program from the LLIR. Furthermore, significant transformations through the application of optimizations on both, the high-level and the low-level representation, could have further changed the layout in considerable ways.

The optimization we attempt to perform is carried out on the ICD LLIR. As we heavily rely on information on the program structure, it is evidend that a way to recover this information has to be found. The goal is to generate an abstract, tree-like view on the program after all other optimizations have been performed.

A popular class of analysis methods is known as *interval analysis*. This kind of program analysis attempts to find and identify well defined structures formed by the nodes of a control flow graph. After identification, the found structure is replaced by a single representing node in the control flow graph. This basic scheme is repeated until the control flow graph has been *reduced* to a trivial graph consisting of just a single node. In fact, what the algorithms are trying to identify is referred to as an *interval*.

Definition 4.2.1. An interval $I(h)$ in a control flow graph is a maximal, single entry subgraph in which h is the only entry to $I(h)$ and all closed paths in $I(h)$ contain h.

This definition implies that an interval can either be a some kind of *cyclic* structure (loops) or *acyclic* structure with both only having a single entry point. We will now examine three

approaches to interval analysis. One of them, called *structural analysis*, was found to be best suited for the optimizations we will perform later.

One of the oldest approaches is the *T1-T2 analysis* [Ull73]. Its basic workflow consists of two steps. The first is the identification of control flow graph nodes forming self-loops and the removal of the back edge. The second step then searches for any pair of nodes with the leading node being the only predecessor of the other and the replacement of the two nodes by a single representing one. During the removal of back edges and the replacement of pairs, a separate data structure can be build that will represent the nesting structure of a function. This approach is not useful for our needs as it only reveals information on loops that it finds in an abitrary order. It only reveals nesting but is not capable of reflecting the actual control flow through different structures of the program in an unambiguous way.

Another classic approach is the interval analysis proposed by [AC76]. Once an interval has been identified, all the nodes that make up this interval are being replaced by a single representing node. The edges that lead into such an interval are corrected accordingly. The result is another valid control flow graph on which the next interval is to be found. During classification of intervals, a data structure will hold information on what nodes of the control flow graph comprised such an interval and what kind of interval was just found. Therefore, such a data structure finally can deliver information on intervals and their nesting structure. The advantage over the *T1-T2* analysis is that the control flow graph is not simply reduced in an arbitrary way but that it is attempted to find maximal intervals. Furthermore, it allows the identification of intervals that cannot be unambiguously identified given the definition of an interval.

Although being more detailed, it still lacks the ability to reveal the actual control flow structure of a program as we can only distinguish loops and non-loops. However, for our problem, it would be advantageous to identify high-level structures within such an analysis so as to avoid doing this later. What we would like to have, is the detection of the exact structures that result from the transformation of the source program (into a low-level representation and the application of optimizations) that are relevant to our own optimization.

4.2.2 Structural analysis

The structural analysis is one of the most powerful analysis techniques as it can identify constructs of a high-level language only by analyzing a low-level representation of a program. It is not only possible to just identify loops and intervals. But it allows to explictly identify what kind of structures are comprised by a set of control flow graph nodes.

An example are loops. While interval-based techniques can only reveal that a loop exists around a certain set of nodes, the structural analysis can distinguish between different kinds of loops. From the C programming language we distinguish two kinds of loops: *while* and *natural* (*do-while*). While the former is a loop whose condition is tested at the top, the latter has its loop condition at the bottom. As we will see later, the distinction between different kinds of loops is inevitable for our optimization. Also, loops can be constructed by using *goto* statements. The structural analysis will identify the loop constructs that is expressed through this, although even the high-level representation would not reveal this. Therefore,

the structural analysis is well suited for our needs as it fully classifies the control flow graph into nested, unambiguously identified structures.

The ideas for this analysis originally stem from [Ros77] but was aimed at code structures typically found in Fortran77 programs. [Muc97] applied the basic concepts to typical representations of the C programming language. Although providing detailed instructions on how the analyis can be constructed technically, a largely simplified input is assumed. Therefore, we extend the proposed algorithm, so that the result fits our requirements regarding the code selector of the WCC.

The basic idea of the structural analysis is not to identify sequences of maximal size as in the interval analysis approaches but to match the current control flow graph against a set of structural patterns. During compilation of a program, each high-level construct is transformed into a low-level representation by generating certain control flow patterns in the code selection phase. This analysis is capable of identifying those patterns and to generate a data structure that will reflect all these structures and their nesting accordingly.

After having identified such a match, the subgraph is replaced by a single representing node. All egdes that originally were connected to any of the subgraph's nodes are reconnected to the newly created node. This process repeats until the control flow is reduced down to a single node.

This process is unambiguous because each reduction only removes a single structure whose possibly nested structures have already been reduced in an earlier step. The result of such an action is again a valid control flow graph. Valid means that the graph resulting from a reduction is equivalent to a graph that would have resulted from removing the identified structure from the high-level representation right away. It literally disappears without affecting other parts of the graph.

Prior to presenting the algorithm, we need to define a few graph properties. A control flow graph of a function only has a single entry node (*source*) and a single exit node (*sink*). In a programming language like C, multiple *return* statements can exist which cause the graph to have multiple sinks. To be able to deal with this, a virtual sink can be attached to the graph to which all sinks are being connected by an edge. For the sake of simplicity we will assume that our graph only has a single sink.

A control flow graph can have back edges. Given the graph would be traversed by using the well known depth first search algorithm, a back edge is an edge whose head node has already been visited but whose tail has not yet been finished. A more formal definition coins the term *dominance*.

Definition 4.2.2. For a graph $G = (V,E)$, the function $dom : V \times V \rightarrow \{0,1\}$ forms a binary relation of nodes $u, v \in V$ so that

$$dom(u,v) = \begin{cases} 1 & \text{if } u \text{ is on every path from the source node to } v. \\ 0 & \text{otherwise.} \end{cases}$$

In other words, u *dominates* v if and only if $dom(u,v) = 1$. This relation is reflexive, transitive and antisymmetric. With this property known, we can define back edges as follows:

Definition 4.2.3. A back edge is an edge $e \in E$ in the control flow graph $G = (V, E)$ whose head dominates its tail.

The structural analysis works correctly on graphs that are *reducible* (or *well-structured*). According to [Muc97]:

Definition 4.2.4. A flowgraph $G = (V, E)$ is reducible if and only if E can be partitioned into disjoint sets of E_F, the *forward* edge set, and E_B, the *backward* edge set, such that (V, E_F) forms a directed acyclic graph in which each node can be reached from the entry node, and the edges in E_B are all back edges as defined in 4.2.3.

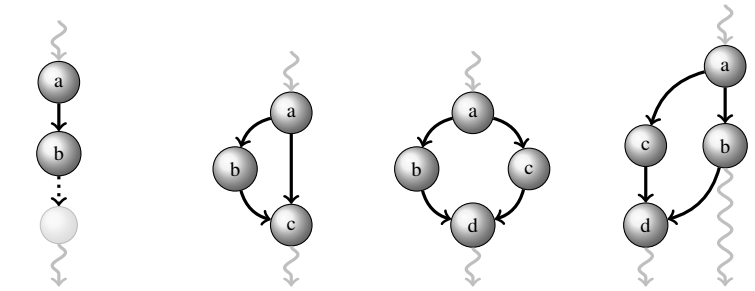
A graph that violates this property is called *irreducible*. Such a graph contains a structure whose dominance property is ambiguous. In a reducible graph, each back edge defines a single loop. The central property as defined by the dominance relation is that loops only have a single entry node. Irreducible graphs are ones that contain loops with multiple entry nodes. In the C programming language it is possible to construct such irreducible graphs by using the *goto* statement to perform jumps into loop bodies. In general, this is considered bad programming practice. Such constructs practically very rarely appear [1] and can usually be easily avoided. Although there are ways to deal with it, the explicit handling is neglected. Only applying the standard language constructs always leads to well-formed control flow graphs.

The algorithm for the structural analysis is now presented. Figures 4.2, 4.1 and 4.3 show the patterns that the analysis is capable to detect. Each one of those patterns has the property that there only exists a single entry into the structure.

In general, two kinds of pattern classes can be distinguished. We refer to patterns not surrounded by a back edge as *acyclic* whereas the ones being surrounded by a back edge are referred to as *cyclic*. The patterns presented here are constructed specifically for the output of the code selector of the WCC. Though, in general, they would fit any low-level representation of a C program.

The acyclic patterns consist of patterns named *block* (4.1a), *ifthen* (4.1b), *ifthenelse* (4.1c) and *proper* (4.1d). A block pattern is a sequence of basic blocks that is entered through the basic block at the top and left through the basic block at the bottom. Thus, no other edges lead into or escape from this sequence otherwise. When a block pattern is encountered it is attempted to maximize it by extending the sequence along the path until the block property as just defined is violated. This pattern is the first one matched against the control flow graph prior to all other patterns. The *ifthen* and the *ifthenelse* pattern match against high-level constructs of the C programming language of the same name. They identify conditional branching in the control flow graph. It is characterstic that both cases branch conditionally and join again in a single node. A special case is the *proper* pattern. It identifies acyclic constructs that have no directly corresponding structure in the C programming language. Its

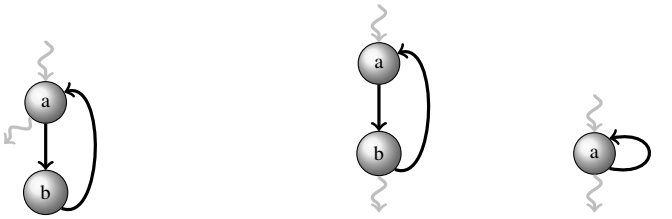
[1] [Muc97] refers to a survey of D.E. Knuth by which over 90% of a selection of Fortran77 programs have reducible graphs. The tests run for this thesis verify this result for C programs. Commonly used constructs that lead to irreducible graphs are Co-Routines [Knu73b], Duff's device [Duf83] and the inconsiderate use of *goto* statements.



(a) *BLOCK* (b) *IFTHEN* (c) *IFTHENELSE* (d) *PROPER*

Figure 4.1: Acyclic patterns

main characteristic is that it branches just like in the *ifthenelse* case but has an edge escaping the structure in one of the branches, thus not having a unqiue point where the control flow joins again. Such a control flow usually appears in C switch statements where the case branches are not separated by a break statement. Switch statements with properly terminated case branches have the form of a continuously nested ifthenelse pattern. Although [Muc97] suggests a separate switch pattern that would catch both the cases just described, it proved completely unsuitable here because it would result in a more complex pattern matching and violates the constraint of atomicity. That is, no pattern contains another one as a sub-pattern.

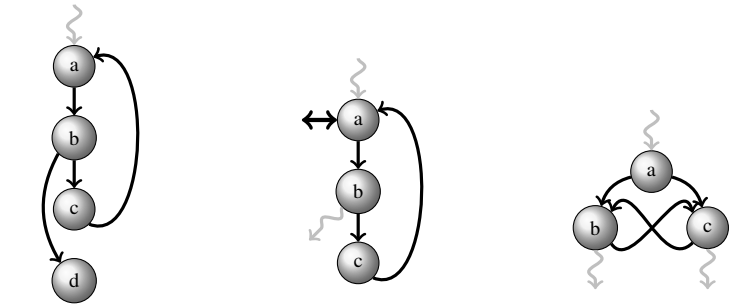


(a) *WHILE LOOP* (b) *NATURAL LOOP* (c) *SELF-LOOP*

Figure 4.2: Cyclic patterns

Besides these, basically three kinds of *cyclic* patterns can be distinguished. The *while* (4.2a) pattern matches any loop that consists of two nodes and a backedge from the bottom node to the top node. In addition, the loop is escaped from the top node by a branch. The bottom node has the back edge as its only outgoing edge. The *natural* (4.2b) pattern matches a loop very similar to the *while* pattern. The difference is that the topmost node does not branch. In opposition, the bottom node does. The latter has two outgoing edges, one leading to another node not inside the match, the other being a back edge to the top node. A third pattern is the *selfloop* (4.2c). It comprises of just a single node reaching itself through a back edge.

With the patterns described so far, most control flow graphs generated from a C language input can be reduced. However, there exist a couple of special cases. The iteration of a loop can be irregularly interrupted by using *break* statements within the C language. Such a



(a) *LOOP BREAK* (b) *WHILE/CALL LOOP* (c) *IMPROPER*

Figure 4.3: Special patterns

statement causes an unconditional and immediate escape from the loop bypassing all regular loop condition testing. The regular test is performed at the top for *while* and at the bottom for *natural* loops. The control flow continues at the node which would have been executed after the regular loop condition would have failed. Therefore, a single *break* statement in a loop would create a pattern as shown in (4.3a). The primary characterstic of this pattern is that there exists a conditional branch within the loop with one edge continuing the loop and the other escaping the loop. The latter edge leads to the node directly following the loop in the control flow. It is noteworthy that the pattern shown is just schematic. In fact, the escape condition can be arbitarily deeply nested within other structures - not being loops or a *switch* statement - within the loop. The strategy to overcome the problem of not knowing the enclosing loop yet when matching such a pattern is to simply discard the escaping edge upon reduction. It is not necessary to verify correctness here because the parsing phase of the compiler would already fail if a *break* statement was used outside a loop (and outside a *switch* statement).

Another problem arises with the use of function calls within the condition expression of a *while* loop. This inevitably leads to the splitting of the loop condition into two separate basic blocks. This is shown in 4.3b. The corresponding pattern is named a *while/call* pattern. Such a situation can unambiguously be identified by the following facts. The topmost node has two incoming edges, of which one is a back edge coming from the bottom node. The node beneath the topmost node is unconditionally reached by the latter and has two outgoing edges. One egde leads into the loop the other escapes the loop to a node outside of it. The fact that neither the top nor the bottom node performs any conditional branching distinguishes this pattern from the *break* pattern in the case of ambiguity. The reduction of a match in the control flow graph consists of just discarding the two topmost nodes. The consequence is that this loop can be matched against a *while* pattern immediately. Figure 4.3c illustrates an example of an irreducible structure.

In figure 4.4, an example is given to show the effect of matching and reduction. Given the control flow graph, the algorithm proceeds from the bottom up. The first match is an *ifthenelse* structure (1). The complete pattern is replaced by a single node, all edges that were connected to nodes within the match are reset to the new node *j*. The search continues

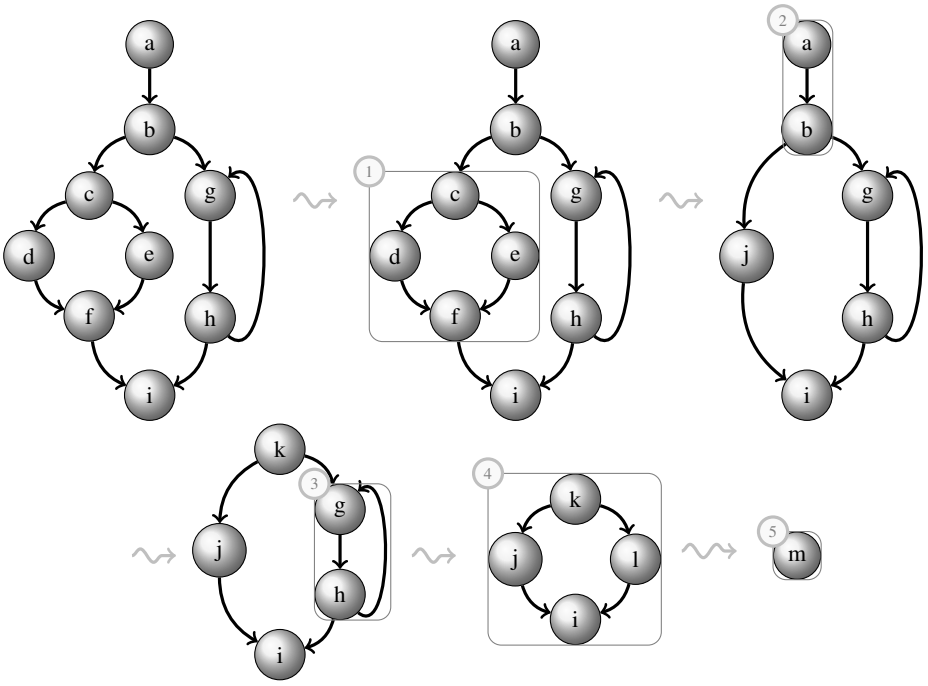
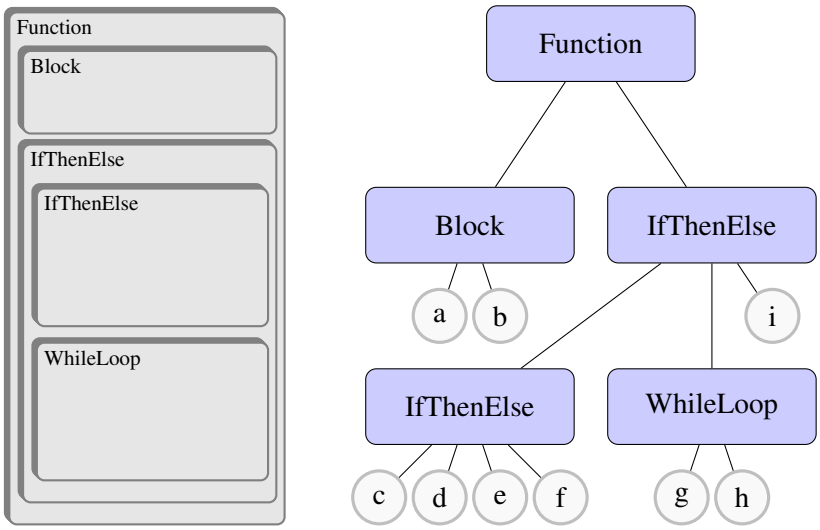


Figure 4.4: Example of structural analysis

and matches a *block* (2). Again, the match is replaced. Since the search reached the topmost node, the process is restarted at the bottom. In (3) a *natural* loop is identified and replaced. The final pattern that matches is a *ifthenelse* structure (4). After handling the latter, only a single node is left (5) and the algorithm terminates.



(a) *Nesting* (b) *Control-tree*

Figure 4.5: Nesting structure and control-tree of example 4.4.

While matching the graph against the given patterns, we need a way to recall what patterns we identified and what nodes were enclosed by them. The data structure to do this is referred to as a *contol tree*. A *contol tree* is a n-ary tree whose root node represents the fully reduced graph and whose leafs represent the basic blocks of the function represented by the control flow graph. Every other node represents a matched and reduced structure. Nesting is expressed by attaching structurally enclosed structures as successors to other nodes. The given example that demonstrates the reduction of the graph in figure 4.5, shows the corresponding control tree that would have been generated by the algorithm.

After having discussed the fundamental ideas of the algorithm, we can now examine how it is actually implemented.

Algorithm 1: Algorithm of structural analysis

```
1:  {cfg is input control flow graph}
2:  v: node
3:  region: set of nodes
4:  depth_first_search(cfg) {determine order of processing}
5:  v := post_order_start(cfg)
6:  while |v| > 1 do
7:      region := find_acyclic_region(v) {see algorithm 2}
8:      if |region| = 0 then
9:          region := find_cyclic_region(v) {see algorithm 3}
10:         if |region| = 0 then
11:             v := post_order_next(v)
12:         else
13:             v := reduce(region)
14:         end if
15:     else
16:         v := reduce(region)
17:     end if
18: end while
```

As already mentioned, a repetitive matching of patterns and the reduction of matches is performed. The actual strategy is to traverse the graph in its reverse topological order. This is achieved by applying a depth first search on the control flow graph. A depth first search can assign numbers to nodes while traversing it. These numbers define the order in which the nodes have been visited. Usually the numbers are assigned in pre-order. This means the numbering is performed when a yet unvisited node is reached. The fact that the control flow graph is matched against the patterns starting with the entry node of each pattern makes such an ordering useless for our needs. When a reduction took place, it is not clear where to continue the search for matches since the patterns can have multiple exiting edges. Therefore, it is more convenient to step through the graph in post-order. This means that the nodes are numbered when a node is completely processed. That ordering implies that the exit node receives the lowest number. Now, when a match was found and a reduction took place, we can simply assign the highest number of a node of a matching subgraph to the replacement

node and continue our matching right at the current node. All in all, we are traversing from the bottom upwards. This is shown in algorithm 1.

After each step, the known patterns are matched against the current node and its succeedors. The acyclic structure are tested with the function *find_acylic_region* in line 7. If this fails the current node is tested for whether is matchs a cyclic structure by the function *find_cylic_region* as shown in line 9. If any pattern matches a subgraph, the latter is discarded from the control flow graph and replaced by a representing node. This is done by the function *reduce* in line 16. All edges that originally lead into the subgraph are reset to be connected to the newly introduced node. The search is then continued with the next node according to the order defined by the post-order depth first search numbering (*post_order_next*). Once the whole control flow graph has been traversed, we start at the bottom once again. Every single pass through the graph detects at least one match.

Algorithm 2: Function to detect acyclic regions

```
 1: {u is input}
 2: region := ∅
 3: v, l, r: node
 4: region := maximize_block(u)
 5: v := head_node(region)
 6: if |region| ≥ 2 then
 7:    insert_ctree_node(v, region, block)
 8:    return region
 9: else
10:    l := left_succ(v), r := right_succ(v)
11:    if out_degree(l) = out_degree(r) = in_degree(l) = in_degree(r) = 1
           ∧ succ(l) = succ(r)
       then
12:       insert_ctree_node(v, region, ifthenelse)
13:       return region
14:    end if
       ⋮
15: end if
```

We will now investigate how the patterns are matched against the graph in detail. The detection of acyclic structures is partly presented in algorithm 2. The function is called with the starting node as its argument. Firstly, it is attempted to find a block of maximal size. As figure 4.1a suggests, a block structure is a sequence of nodes only entered at the top node and only left from the bottom node. The function *maximize_block* in line 4 first searches along the control flow until a node is encountered that violates the block property. Next a search from the starting node upwards is performed. The node v represents the head of the block afterwards. If a block structure has been detected it consists of at least 2 blocks (line 6). If this is true, the function *insert_ctree_node* generates a new node in the control tree. Since the pattern detection works from the most deeply nested structures upwards, the control tree is constructed bottom-up. Since we can easily determine which existing nodes

are contained in a structure by looking at their corresponding control flow graph nodes, the control tree can conveniently by constructed. The arguments to the function are the leading node of the match (according to the control flow direction), the complete set of control flow nodes that comprise the match and the type of the match. After creating such a control tree node, the set of nodes comprising the structure is returned so that it can be reduced.

If no *block* has been found, the adjacent nodes of the head node are examined to determine other acyclic patterns. In line 10 of algorithm 2, an example of how to detect a *ifthenelse* structure is shown. The test is straight forward. If the subgraph matches, again a control tree node is generated like explained earlier and the set of nodes is returned.

Algorithm 3: Function to detect cyclic regions

```
 1: {u is input}
 2: region := u ∪ {v ∈ V | ∃w ∈ V : has_fp(v,w) ∧ !has_fp(u,v) ∧ has_bp(w,u)}
 3: if |region| = 1 then
 4:     insert_ctree_node(n, region, selfloop)
 5:     return region
 6: end if
 7: if |region| = 2 then
 8:     v := region \ u
 9:     if in_degree(u) = 2 ∧ out_degree(u) = 2 ∧
        in_degree(v) = 2 ∧ out_degree(v) = 2 then
10:         insert_ctree_node(n, region, while)
11:         return region
12:     end if
        ⋮
13: end if
```

Given that all tests for acyclic patterns failed, the function *find_cylic_region* as shown in algorithm 3 is invoked. Again, the input is a node of the control flow graph. To detect whether the argument node is the head of a cyclic pattern and what nodes are enclosed in this structure we perform the test in line 2. A cyclic structure contains node u and it contains any node v for which there exists a possibly empty path to a node w (the bottom node of a possible loop) so that u does not lie on the forward path from v to w but there does exist a back path from w to the head node u. All nodes v of the control flow graph that fullfil this property are enclosed in some cyclic structure of which u is its topmost node. If the set of nodes just determined has the size of 2 we can match it against the loop patterns presented above. The handling of the structure after detection is equivalent to the actions performed when an acyclic structure is found.

The structural analysis is a powerful tool when it comes to reconstruction of typical control flow structures. The advantage over methods that only partition the graph into acyclic and cyclic structurs is that a full identification is also performed. The creation of a control tree that fully models a function's control flow simplifies any optimization that relies on modification of only well defined parts of a program. The algorithm presented here is only a rough outline of the steps necessary to perform. In [Muc97], it is presented in much

more detail. However, the description there is still incomplete. It is noteworthy that the structural patterns as shown above might need additional modifications to suit the low-level presentation generated by other code selectors. The ones here are set up for the WCC.

4.3 Towards a static optimization

Program code has the property that it is in general not altered at compile time. Also jump targets are usually decided at compile time. Therefore, a static model of a program's control flow can be established. It is understood that for modeling a globally complete control flow, all functions involved, and their local control flow need to be known. The proposed optimization only needs to deal with local control flows. Although also here the complete set of called functions needs to be known, it is not necessary to know the global control flow between functions. Such a control flow would reveal the order in which function invocations would appear. This information is not required for this static optimization.

That said, the strategy of this optimization is to select subsets of a program at compile time so as to achieve an optimal reduction of the WCET given a scratchpad memory of certain size. The static assignment to a scratchpad memory resembles the Knapsack Problem ([Weg99]).

Definition 4.3.1. Knapsack Problem (KP)
Given a knapsack with a weight limit of W and n objects $o_i \in O$. The function $w : O \rightarrow \mathbb{N}$ denotes the weight, the function $v : O \rightarrow \mathbb{N}$ the *usefulness* of an object.
Find a subset $O_{opt} \subseteq O$ so that $\sum_{o_i \in O} v(o_i)$ is maximized while the weight limit is not exceeded. That is $\sum_{o_i \in O} w(o_i) \leq W$.

In our case the knapsack is the scratchpad memory and the objects would be any type of program *code objects* like functions, basic blocks and instructions. The weight of an object is the accumulated size of the objects. The *usefulness* is the *gain* we can achieve by moving such an object from the main memory into the scratchpad memory. Such gain can be the reduction of execution time. In our case the aim is to reduce the upper worst-case execution time bound.

Moving single instructions makes little sense since the the different memories are located in different places in the address space and jumps would be necessary to reach each one of them. Assigning basic blocks to different memories makes use of the fact that such a block only has a single entry point at the top and a single exit at the bottom of a sequence of instructions (see definition 2.2.1). This implies that a block either ends with an implicit or an explicit jump. The latter being conditional or unconditional. A basic block has no side-effects on the control flow. Therefore, only a block's tail would require modifications to correct the execution path so that a program would run utilizing both memories. We will investigate this issue later.

Moving a basic block into the faster memory just because its isolated gain is high does not consider the overhead caused by jumps between memories. Given that a sequence of basic blocks is connected solely by implicit jumps, it would be comparably expensive to move some of them into a different memory. This would require the introductio of additional jump

instructions which have negative effects on the execution performance. Because of this, many optimizations proposed so far attempt to form groups of basic blocks to achieve better results. Typically, sets of basic blocks that are executed in forward direction (in the direction of the control flow) are chosen. Loop bodies are such sequences. As long as the graph is reducible as defined in 4.2.4, only loops with a single entry node and a single successor node are encountered. Such sequences are also referred to as *traces* (see [VWM04b]). Since the code in a loop is executed repeatedly, moving it into a faster memory alltogether potentially would yield the best results.

Another option is to move entire functions. This comes with two problems. From the function entry many different execution paths may lead through it. Some of which are rarely exectuted or result in a low gain when moved into a faster memory. Picking whole functions can be too coarse grained, and space occupied by rarely executed instructions is better off used for code that would increase the gain.

Although manifold code optimizations utilizing scratchpad memories exist - some of which have been mentioned above - they are all unsuitable for the problem of optimizing the WCET. The reason for this is that all proposed optimizations need not consider a worst-case time bound. They have in common that most expensive parts of a program are picked and moved into a different memory. For energy reduction the expected energy dissipation is considered. For optimizations concerning the average execution time, the expected execution of each such fragment is considered. This kind of average case optimization can be precisely mapped onto the KP as described above. Compared to that, WCET-centric optimizations need to actually know the upper execution time bound. As has been described in chapter 2, this bound can be calculated either implictly using *IPET* or by explicitly searching the worst-case execution path (WCEP). Since our aim is to optimize the worst-case bound we need to model this path in some way, so that optimization decisions are correct. Those decisions are the allocation decisions on program code. The fact that this path is potentially unknown prior to evaluating the problem, makes the selection of traces an unsuitable method. Such a selection is solely based on the assumption that an expensive trace ought to be allocated in the scratchpad memory irrespective the fact that such a trace might not be part of the actual WCEP. Therefore, this static precomputation would not lead to an optimal result.

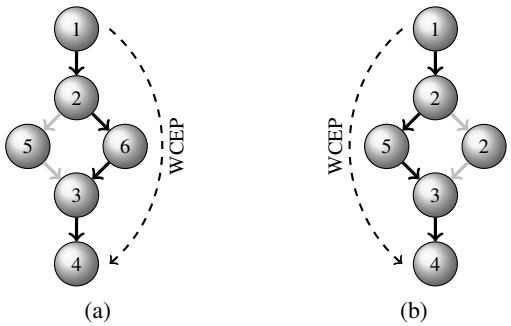


Figure 4.6: Dynamically changing WCEP during optimization

In figure 4.6a, an example path through a program with a marked WCEP is illustrated. The nodes are labeled with a fictive WCET. If an optimization is applied that leads to a change in any of the separate WCET, the WCEP can change dynamically as shown in 4.6b. This is the key problem for all WCET-directed optimizations. A modification during an optimization potentially leads to a change of the WCEP. For techniques making use of explicit path information, a repetitive reevaluation of this path is inevitable.

As has already been pointed out, single basic blocks might be too fine grained to justify their isolated allocation. Given, we ignore this problem for a moment, another major issue comes up. Whatever selection of basic blocks has been performed, as the program does usually not fit into the faster memory as a whole, some modifications to program code need to be done, so that the control flow remains equivalent to the unoptimized program. For basic blocks that are executed sequentially through an implicit jump, an explicit jump would need to be generated if those were distributed among the memories. Depending on the architecture, multiple additional instructions might be required. For implicit jumps such correction inevitably leads to a growth of program size. But also for explicit jumps cases, this might be the case. RISC architectures are especially prone to this due to the limited size of their operands. To give an example, the TriCore1 architecture assigns address intervals of 256MB to each memory. There exists just a single unconditional jump instruction which takes an immediate, signed 24bit address operand. This means that at most a range of 8MB is immediately addressible. Other jump instructions are even less powerful. The only way to achieve such long distance jumps is to jump indirectly by loading the jump address into a separate register and execute an indirect jump instruction. That way, the result is at least one additional instruction. And the case just described is only one of the more trivial cases. This issue is thoroughly discussed in section 4.5.2. Either way, the inflation of the program due to such an optimization of the code is difficult to track precisely. The inflation occurs upon deciding for a certain allocation and has immediate impact on the blocks involved and the adjacent ones. This is where caches have a clear advantage, as has already been pointed out. They are fully transparent to the instruction fetch unit, thus requiring no additional modifications. It is remarkable that this issue hasn't been addressed yet in any optimization of program code for scratchpad memories - whether they were related to energy reduction or were WCET-centered. In conclusion, the problem we are going to solve cannot be reduced to the KP anymore. Decision on the objects have impact on both measures, weight and usefulness. In the upcoming sections, a solution will be presented.

The WCEP consists of the basic blocks that make up the WCET. To identify the WCEP, two basic approaches are known. In [Pua06], a repetitive reevaluation of the WCEP is performed. A certain number of allocation decisions is allowed, then a complete WCET analysis is performed and the most expensive path is explicitly determined. This approach comes with two drawbacks. On the one hand, just a single allocation decision for a basic block (or any other grouping that was chosen) can lead to a change of the WCEP. The path along which the optimization is performed might not be the most expensive one anymore. Still, further decisions are made. These decision are never cancelled even if that leads to suboptimal results. On the other hand, the repetitive evaluation comes at the cost of long solving times.

The second approach is the one presented in [SMRC05]. Their aim is to optimize program

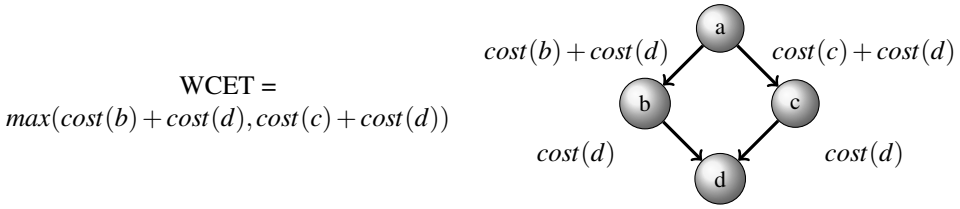


Figure 4.7: WCEP property: Accumulating WCET

data. But the basic idea can also be applied to program code. It is an implicit enumeration problem. Just like with IPET, the attempt is not to explicitly discover paths through the control flow graph but to implicitly take the longest path into consideration. The key to solving the allocation problem for program code is to take the worst-case path into consideration on each single decision and to not decide for allocation if this proves suboptimal in any way. Because of this, constructing an ILP is the favorable way to get a solution. The central discovery presented in the paper referenced above is that the total WCET along the WCEP is the sum of all WCET of its components. Let's consider just basic blocks and a trivial control flow without loops as presented in figure 4.7.

The WCET from the bottom node to the top node is equal to the WCET from the bottom to the node following the top node plus the WCET of the top node itself. Because of this, attempts to reduce the WCET have to be performed on this path. If there are two paths through that graph, then an improvement can only be achieved by making decisions on that particular path which is the WCEP, given the current allocation decisions. This idea is one fundament of the optimizations that are presented in this thesis.

In the following, an ILP for the static allocation of a program to the scratchpad memory is presented. The ideas which have been discussed only briefly above, will be presented in all detail.

4.4 ILP Model

We will now discuss how an actual ILP can be constructed to solve the static allocation problem. The conclusion from the previous section is taken as a reference. The section is split into two parts, a formal description of the problem in section 4.4.1 and the transformation into the final ILP in section 4.4.2.

4.4.1 Preliminaries

In this section, the formal requirements to construct an ILP model to optimally solve the static, WCET-centric code allocation problem is presented. As the name suggests, we try to decide for whether certain parts of a program shall be statically allocated to the scratchpad memory.

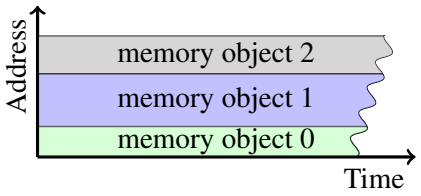


Figure 4.8: Memory layout of static allocation

Before we construct the ILP model, certain preconditions have to be recorded. The allocation decisions and the actual placement will be made at compile time. This means that blocks assigned to a certain memory will occupy it throughout the program's lifetime. This means that during runtime, there is no change - neither in the main memory nor in the scratchpad memory. No two memory objects will be assigned to memory so that they overlap. In turn, this implies that these objects will require a linear layout in the two memories as outlined in figure 4.8.

Although assigned to two different memories, the objects will keep a partial order. The result is that the direction of control flow remains the same after optimization. This means that assumptions made about the machine state[2]remain largely valid[3].

Each control flow graph from which we will generate the ILP model represents a single function. An example is given in figure 4.9. As was already mentioned earlier on, global flow information needs only be available to model whether if and who calls another function. The order in which this happens is irrelevant. Because of this, we can restrict ourselves to just investigate *local* control flow graphs $G_f = (V,E)$. However, we need to be able to construct such a graph for all functions involved. Since the optimization takes place as an integrated step in the WCC compiler, the complete program needs to be provided as its input. Figure 4.9(1) shows the control flow graph that would result from the given source code.

Such a graph needs to be reducible as defined in 4.2.4. Reducibility requires that the graph can be partitioned into two sets. The set of forward edges E_F and the set of backward edges E_B. Discarding the backward edges from the graph therefore leaves us with a *directed acylic graph* $G_{DAG} = (V, E \setminus E_B)$. This graph has a single entry node $v_s \in V$ (source) and a single exit node[4] $v_t \in V$ (sink). Trivially, every path from v_s to v_t contains no cycles. Applied to the concrete program, this means that all loops have been discarded in G_{DAG}, leaving only the loop bodies. This is outlined in 4.9(2). In the figure, when the back edge (e,c) of the loop is discarded a DAG results. In addition a virtual sink node f has been introduced so that this graph now has a unique source and a unique sink.

Loops themselves have properties comparable to local control flow graphs. An illustration if its properties is given in figure 4.10. Given that $V_L \subseteq V$ is the set of nodes comprising

[2]In our case the state of the TriCore1 pipeline and the outcome of its static branch prediction.

[3]The corrections on the program code to reconstruct the control flow between the different memories can also impact the execution significantly.

[4]For a control graph with multiple sinks a virtual sink node is introduced prior to construction, to which all the previous sinks are connected.

```
int x;
x = n + 1;
if (x < m)
  return m;
else
  do {
    x += m;
  } while(x < m*m);
return x;
```

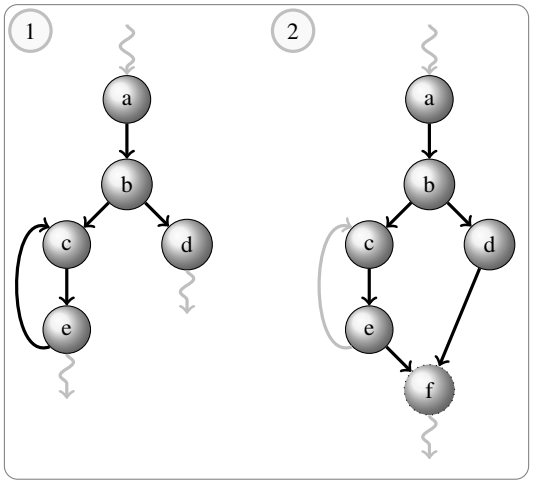
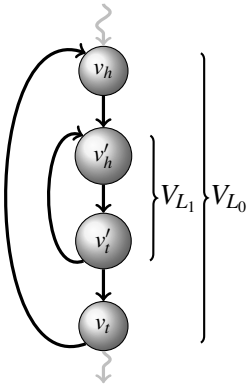


Figure 4.9: Local control flow graph properties



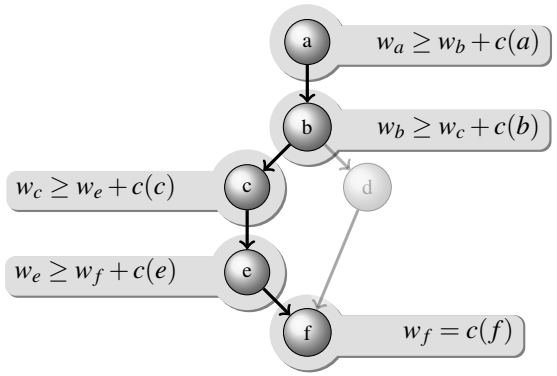
$$G = (V, E)$$
$$V_{L_1} = \{v'_h, v'_t\} \subset V_{L_0} = \{v_h, v'_h, v'_t, v_t\} \subset V$$
$$E_B = E \setminus E_F = \{(v'_t, v'_h), (v_t, v_h)\}$$

Figure 4.10: Loop properties

an abritrary loop within G. Every loop possesses a single source node $v_h \in V_L$ referred to as the *loop head*, and a single sink node $v_t \in V_L$ referred to as the *loop tail*. The source node v_h is the head of the back edge enclosing the set of loop nodes, the sink node v_t is the bottom. Therefore, $(v_t, v_h) \in E_B$. All nodes $v \in V_L$ will be referred to as the *loop body*. Clearly, for nested loops, the loop body of a nested loop is a subset of the nodes comprising any enclosing loop.

We strive for the minimization of the overall execution time along the WCEP. With a WCET analysis step we are able to obtain the worst-case execution cycles (WCEC) of each basic block. To decide whether a block is to be placed in the main memory or in the scratchpad memory, the difference of the execution times of the two memories need to be determined before we start to construct the ILP model.

As was briefly outlined in figure 4.7, the WCET is composed of the set of nodes (that represent the basic blocks of a function) which accumulate to the highest WCET value. This set is the WCEP. As we need to avoid to explicitly look up the path this must be encoded



(a) Linear

Figure 4.11: Graph dependencies

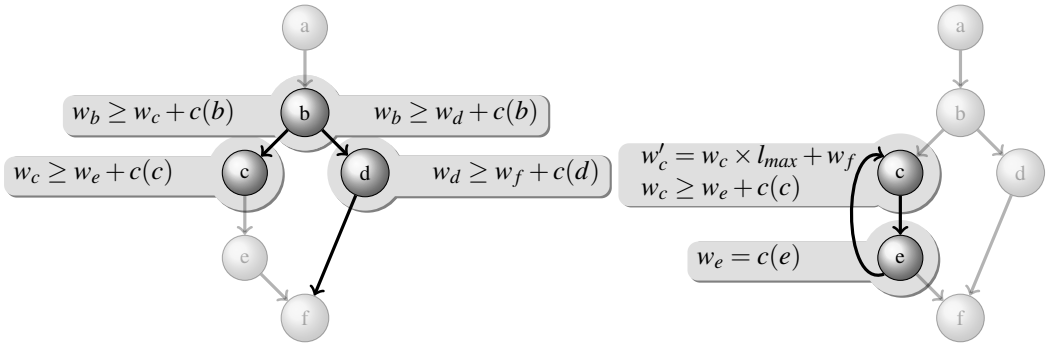
in the ILP model implicitly. To keep things simple at this point, we assume a control flow graph without loops and without branches as shown in figure 4.11a. Since node f is the sink of the graph, the WCEP starting from f only consists of this node. The total WCET is just the WCET of its corresponding basic block. The WCEP starting from node e consists of nodes e and f. Trivially, the WCET is the sum of both per-block WCETs. Put differently, the following general observation can be made:

Theorem 4.4.1. *Given a WCEP P in a control flow graph $G = (V,E)$ that is composed of nodes $P = (v_1,...,v_n) \subseteq V$. Let w_i be the cost of the WCEP from v_i to v_n. That is, the WCET. Let $c : V \to \mathbb{N}$ with $c(v_i)$ reflect the WCET of the single basic block corresponding to node v_i. Then for any $(v_i,...,v_n) \in P' \subseteq P$ the following inequation holds true:*

$$w_i \geq w_{i+1} + c(v_i) \tag{4.1}$$

This central observation is illustrated as a whole in figure 4.11a. As can be seen, a possible path has already been marked which is assumed to be the WCEP. In such a linear dependency, one node's WCET variable w_i depends solely on the succeeding node. Since nodes can not have a negative WCET $c(i)$, the WCET variable that denotes the cost of the path from a particular node down to the sink increases monotonously. Since the aim is to find a tight WCET, the solution of these inequalities should be minimized. This implies that the source node's WCET variable represents the highest accumulated costs, which is the overall WCET bound. The WCET value of the bottommost node is simply its own WCET.

Of course, this presumes that the WCEP is actually known. There can be numerous paths through a control flow graph from its sink to its source. Every branch results in an alternative. Among all paths, the worst-case path is the one whose overall costs are maximal. To solve this problem, we take a step back from the details for a moment and reconsider the overall aim. The aim is to minimize the overall worst-case time. This implies that we have to optimize along the worst-case path. If we encounter a branch, this path can trivially only



(b) Branches (c) Loops

Figure 4.11: Graph dependencies (continued)

lead along one of the branch edges and the path resulting in the maximum execution time must pass along one of them.

Theorem 4.4.2. *Let $G = (E,V)$ be an acyclic control flow graph. Let $outdeg : V \rightarrow \mathbb{N}_0$ denote the number of outgoing edges from a node. Let $v_i \in V$ be an arbitary node for which $outdeg(v_i) = 2$ holds true and which is known to be on the WCEP. That is, $v_i \in P$ as defined in theorem 4.4.1. Also let $v_j \in V$ and $v_k \in V$ be the corresponding adjacent nodes. Then $v_j \in P$ holds true if and only if $max(w_j, w_k) = w_j$.*

To make this more applicable to ILP, we express the observation as inequalities. The WCET w_i at node $v_i \in V$ as just defined fulfills

$$\begin{aligned} w_i &\geq w_j + c(v_i) \\ w_i &\geq w_k + c(v_i) \end{aligned} \tag{4.2}$$

We are now only examining the single branch as shown in figure 4.11b. As can be seen, w_c and w_d are costs of the partial paths from nodes c and d to f, which is the sink of this graph. The tightest worst-case execution time bound at node b can trivially only be the maximum of any of the two worst-case time bounds increased by the cost caused by b itself.

This can be applied to all branches of a graph. Taking into account the equations for simple linear dependencies and the rules for branching in an acyclic graph, the solutions for WCET variables w_i at all nodes represent some worst-case time bound for every possible path or subpath. Solving the equations with the aim to minimize the bound, the entry node's variable represents the WCET.

Until here, we have only dealt with acyclic graphs. A cyclic graph is one which contains loops. Loops are subgraphs whose entry and exit nodes are connected by a back edge. Every entry node may have only single incoming back edge. As we will see later, this is a

precondition to deal with nested loops unambiguously. This restriction implies that every loop can be identified by its entry node. In other words, no two loops share the same entry node. As was already mentioned, the graphs we are dealing with need to be reducible as defined in 4.2.4. In particular, multiple entries into a loop are not allowed.

Loops induce repetitive execution of the loop body. If we assume for the moment that there exists a single loop in an anotherwise acyclic control flow graph, we can observe that the contribution of such a loop to the overall (global) WCET is exactly the local WCET determined by the WCEP through the loop body times the repetition count of the loop. This local WCEP is restricted by the entry and exit nodes of the loop.

Some important facts about loops should be pointed out here. A cyclic region in a control flow graph does not need to exit at its tail node. The tail node is the one node that is dominated by all other nodes within the loop body and from which a back edge orginates. However, as we have seen above, we know at least two kinds of loops, namely *while* and *natural*. The former exits at its top node, the latter at its tail node. As has been shown there, also cases where a loop is exited not at these nodes can occur. We will refer to this as an *irregular* exit. Such an irregular exit interrupts the control flow prematurly. Both paths from such an exit are guaranteed to join right after the loop in the direction of the control flow.

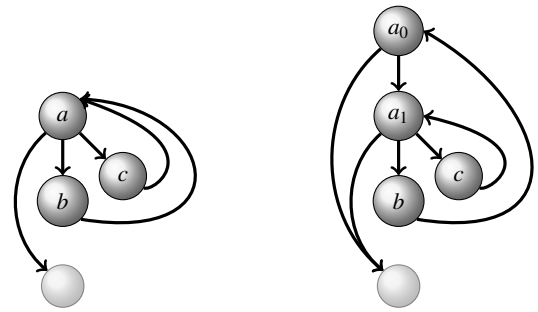
Theorem 4.4.3. *Let $G = (E,V)$ be a cyclic control flow graph with $V_L \subset V$ being the set of nodes comprising the body of a single loop not including any nested loops with a maximum repetition count of l_{max}. Let $v_h, v_t \in V_L$ be the head respectively the tail of the loop according to the loop's back edge as defined in 4.2.3. Moreover let $P \subseteq V$ denote the WCEP of G and let T_G be its corresponding WCET. Then there exists some path $P' = (v_h, \ldots, v_t) \subset P$ whose local WCET T_L from v_h to v_t contributes to T_G by exactly $T_L \times l_{max}$.*

Put differently, the global WCET w'_h of the WCEP starting from the loop's head node v_h and whose succeeding node be v_s with its accompanying costs w_s, fulfills the inequation

$$w'_h \geq w_h \times l_{max} + w_s \tag{4.3}$$

If we already knew the loop's local WCET denoted by w_h, any predecessing node could depend on it to express its own WCET inequation just as described earlier on. If a loop has been specified in terms of equations as in equation 4.3, it can be treated as if it would not have existed in the graph, but instead would be represented by a single node with a single WCET value, equal to that of the whole loop. This is because the loop is entered and repeated at unique nodes. All depending equations can be set up accordingly.

An example of a natural loop in figure 4.11c. The variable w_c represents the WCET from node c along the WCEP down to node e, which is the tail node of the loop body. As can be seen, node e does not depend on any other node in the graph. This is because the loop body can be treated as an isolated set of nodes, just as theorem 4.4.3 suggests. The loop itself has no nested loops. In this case the loop body forms a trivial linear dependency between just the two nodes. Setting up the equations for the loop body according to theorem 4.4.1, the solution with the minimal value for w_c reflects the total costs of the WCEP within that loop. To take that loop into account globally, w_c is multiplied by the maximal iteration count of



(a) Considered a single loop (b) Unambiguously two loops

Figure 4.12: Ambiguity of loop heads

this loop. A new variable w'_c is introduced which corresponds exactly to the WCET orginating from the complete loop. Any predeccessing node needs to refer to this representative variable instead of the WCET variable w_c itself. A dependence to the succeeding nodes has to be established as well. The global WCET of the path starting at node c including the loop is therefore represented by the variable w'_c which depends on both the loop it represents and the loop's succeeding node.

Naturally, structures like linear sequences of nodes, branches and loops can be arbitrarily nested. Especially for nested loops, each loop must be identifiable separately to correctly construct the inequations. This is an important justification for the restriction that no two loops share a common entry node. Consider the situation in figure 4.12. The problem here is that we can identify two different sets of nested structures. Figure 4.12a could be either two loops sharing a single head or one loop containing a branching control flow as its loop body. Due to the restriction that potential loop head nodes identify only a single loop, the two loops are only identified if the graph has the structure shown in 4.12b.

With this knowledge about the construction of depedencies to obtain the WCET of a control flow graph of a function, we can now fully describe what is necessary to solve the originally stated problem. The problem was to decide for a set of basic blocks which to allocate into the scratchpad memory. We exploit the knowledge about the structural dependencies we just obtained. As was shown, the WCET itself can be obtained from the entry node of the function. It is a solution to a system of inequalities. Specifically, it is the smallest valid solution - therefore representing the tightest worst-case time bound.

4.4.2 Construction

On the following pages, the construction of an ILP is described in all detail. It is investigated how we can use the fundamental dependency equations presented above to solve the static allocation problem.

The set of equations to obtain the WCET along the implicitly discovered WCEP does not yet lead to a solution we require. Yet, the equations model dependencies among connected

nodes. If we want to achieve any allocation decisions, additional effort is required.

An ILP has the general form of a single *objective* function and a set of equalities or inequalities referred to as *constraints* which are used to limit value ranges of variables that the objective function depends on. As the name suggests, the variables in the system are integral and all equations are linear. The objective is to either minimize or to maximize according to a given function. Applied to what has been said in the previous section, we can already construct an ILP to obtain the WCET given a control flow graph and the knowledge about the WCET of all single basic blocks. The equations are created exactly in the way described in the theorems 4.4.1, 4.4.2 and 4.4.3. The overall objective is to minimize the WCET variable of the first basic block of a function.

The optimization we are trying to build should optimize along WCEP. To achieve any kind of decision along this implicit path, we must allow the constraints to lower the bound they represent. A WCET variable per node represents the WCET from that node down to the exit node of the control flow graph. Such a constraint elevates the bound it represents by the costs its corresponding basic block causes. Investigating a constraint that is part of the WCEP, lowering that cost would potentially reduce the costs of all the WCET variables of all nodes before (in direction of the control flow) the current node. Minimizing the value of the WCET variable that represents the source node can therefore be achieved by lowering the costs of any node along the WCEP. Lowering the costs of a node not being on the WCEP has no effect on the overall costs at the source node. This observation is the key to an optimization along an implicitly given WCEP.

Basic constraints

To make use of this knowledge, an additional variable is introduced to the discussed equations. This is referred to as a *binary decision variable*. We will refer to the main memory as mem_{main} and to the scratchpad memory as mem_{spm}. For every basic block represented in the control flow graph node v_i, there exists such a variable a_i defined as:

$$a_i = \begin{cases} 1 & \text{if basic block of } v_i \text{ is assigned to } mem_{spm} \\ 0 & \text{otherwise} \end{cases}$$

Through a_i, we can impose dynamic behavior within the ILP. The assignment of basic blocks to different memories results in change of the per-block WCET. To know both the timings for each memory, the whole program needs to be WCET-analyzed with all its basic blocks assigned to mem_{main} on the one hand and mem_{spm} on the other hand. This way static information is available that can be used as constants within the ILP. We define two functions that will map nodes to the basic block's respective WCET. Let V be the nodes of the control flow graph. Then the two cost functions

$$c_{main} : V \rightarrow \mathbb{N}$$
$$c_{spm} : V \rightarrow \mathbb{N}$$

are defined as

$$c_{main}(v_i) = \text{WCET of basic block of } v_i \text{ in } mem_{main} \tag{4.4}$$

$$c_{spm}(v_i) = \text{WCET of basic block of } v_i \text{ in } mem_{spm} \tag{4.5}$$

They represent the costs imposed by assigning a basic block to the respective memory.

With this information available, we can express how the overall WCET is altered by assigning basic blocks to different memories. By assigning a block to mem_{spm}, we can achieve a *gain* concerning its per-block WCET. This gain is the difference of the result of c_{main} and c_{spm} given that the basic block has been assigned to mem_{spm}. We define the gain as:

$$gain(v_i) = c_{main}(v_i) - c_{spm}(v_i) \times a_i \qquad \forall v_i \in V \tag{4.6}$$

To minimize the overall WCET from the given equation model we have defined above, we incorporate equation (4.6). To achieve this mimization, the static costs used in the latter model are exchanged by the gain formular as just defined. The result is that picking a value for a_i will now dynamically have impact on the overvall WCET.

Let v_0 be the source node of the control flow graph and v_t the sink node. Then in a linear sequence of nodes $P = (v_0, ..., v_t)$, the follwing equations model their costs:

$$w_i \geq w_{i+1} + c_{main}(v_i) - gain(v_i) \times a_i \qquad \forall \{i \mid v_i \in P \setminus \{v_t\}\} \tag{4.7}$$

and

$$w_t = c_{main}(v_t) - gain(v_t) \times a_t \tag{4.8}$$

In this context, a linear sequence is one where the number of incoming and outgoing edges on each node is equal to 1.

For branches, the set of rules is extended for each of the succeeding nodes as defined in theorem 4.4.2. The general form is equivalent to the one just presented.

Loops have been discussed briefly. Theorem 4.4.3 suggests that a loop consists of a loop body and with a unique loop head. The equations defined so far can be applied inductively. That way nested loops can be covered. Otherwise, a loop body only consists of linear sequences of nodes or branches. To represent the costs a loop structure can cause in the overall model, a representing WCET variable is introduced that reflects that costs of the loop body times the maximal loop iterations possible. Every preceeding node needs to refer to this newly introduced WCET variable and not to the variable of the loop body's entry node. An exception are self-loops where the loop body consists only of the loop head itself. The costs are determined similarly to the other two loop constructs.

This model however is simplistic. In fact, loop structures in a control flow graph always have an exiting edge. Depending on the type of the loop construct (refer to section 4.2.2), this edge can be either at the top or the bottom of the loop body. In addition, loops can have irregular exiting edges that can be arbitrarily distributed within the loop body. To make it applicable to ILP, this has to be considered.

The constraint representing a whole loop also needs to depend on the node that immediately follows it, irrespective of the loop type. No constraint corresponding to a node of the control flow graph may refer to a constraint outside the loop body. Only the constraint that represents the overall loop costs may do so.

Without discussing the issue explicitly, the situation of a *natural* loop has already been presented in 4.11c. As can be seen, the costs of the overall loop are solely determined by a single constraint. The fact that node *e* has an out-degree of 2 is ignored. This property is decisive for the ability to treat any structure as if it only consisted of a single node with the accumulated costs of the structures it represents. *Self-loops* are treated similarly.

For a loop without an irregular exit, no changes have to be applied to any constraints apart from what has just been described. Dependencies are only modeled by the constraint representing the full loop. As just mentioned, no loop body constraint may reference other constraint outside the loop. In the case of irregular exits, the execution of a single iteration is interrupted prematurly. The aim of our optimization is to decrease the worst-case execution time along the WCEP. A premature exit edge can trivially never be part of the WCEP because all paths through the loop body all join at a single common successor node of the loop. Such an exit firstly can stop the loop body from executing on any iteration. Therefore, at least one more iteration would still be performed regularly. Even if that exit happens to be in the final iteration, there is still a non-empty path in the loop body at least down to its bottom. Because of this observation, we can simply omit premature exits from a loop when creating the ILP constraints. The worst case will never be affected by this.

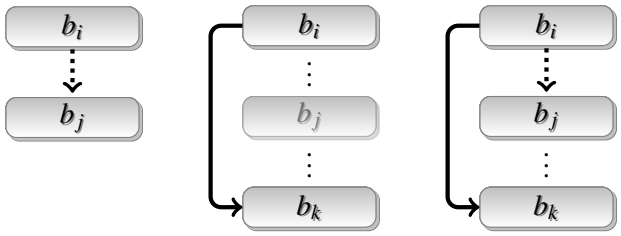
Selection of groups

Up to this point, we have constructed a simple model to obtain the WCET of a function. This is done by creating constraints that model the dependencies among nodes of the control flow graph. The decisions have been constrained to the capacity of the scratchpad memory. This exactly matches the KP. Apart from their dependence according to the control flow, the model does not imply any other dependencies among those blocks. As we have discussed in the introduction, moving small units can be costly. This is due to the fact that the instruction pointer needs to switch back and forth between the two memories which are mapped into the address space at different locations. Since the program formerly remained within the same memory, jumps with only short distances are required.

Distributing basic blocks arbitarily would result in a large number of possible modifications to the instructions code.

A technique is required to keep blocks from being abitrarily distributed solely based on their own WCET. Since we do not know the WCEP upfront, a solution has to be integrated into the ILP. This would lead to a solution where dynamic decisions on the allocation have an immediate impact on the overall costs due to the overhead caused by suboptimal jumping. The only way to keep this overhead low is to form groups of basic blocks and move them into the other memory as is.

Modifying instructions usually means to replace jumps that are unsuited for bridging long



(a) Implicit (b) Unconditional (c) Conditional

Figure 4.13: Typical jump scenarios on RISC

distances in address space by more powerful ones. Especially on a RISC architecture this is often the case. The specific issues related to the TriCore1 architecture will be handled below. To minimize the overhead, the number of jumps needs to be reduced. It must be reduced as long as the advantage of moving a single block into the scratchpad memory does not outweight the overhead. Otherwise a basic block should be kept in main memory.

On a typical RISC architecture three different cases can be distinguished.

- **Implicit**: Two consecutive basic blocks in control flow and adjacent in address space and whose top block does not contain a jump instruction (figure 4.13a).
- **Unconditional**: Two consecutive basic blocks in control flow where the top block contains an unconditional jump instruction to reach the other (figure 4.13b).
- **Conditional**: Three basic blocks of which one is reached from the top block by an implicit jump and the other is reached by a conditional jump instruction (figure 4.13c).

Every jump scenario, as shown in the figure 4.13, is one which can be encountered on every single block. The aim should be to keep any modifications at a minimium. Note that we do not yet consider function calls. That is, control flow among different control flow graphs.

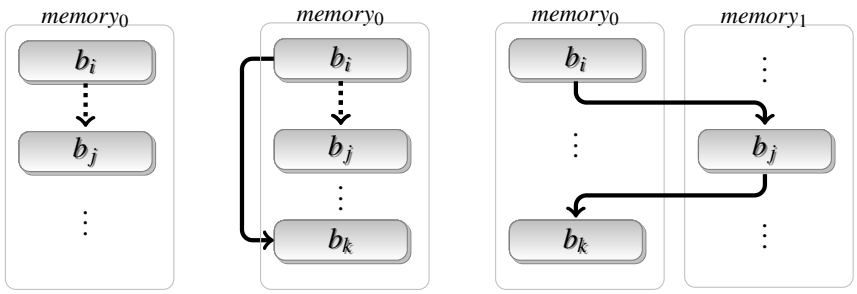
The problem until here has been described rather sketchy. We should now strive for tangible definitions. The overall aim is to minimize the value of the WCET variable of the control flow graph entry node. Any allocation decision does lead to an overhead. This cannot be avoided as a whole because we need to distribute our program among the memories to achieve our goal. However, the gain must always outweight the overhead. In addition, we would like to make decisions for larger groups of nodes at a time. To model this overhead we introduce a new function

$$c_{jump} : V \rightarrow \mathbb{N}_0 \tag{4.9}$$

that will model the jump costs and simply add its mapped value to every per-node constraint.

The function must properly reflect what situation happened to exist prior to allocation and what immediate costs some allocation decision would have on a particular basic block. We refer to this overhead as the *jump penalty*. The function will be defined after we have made clear how such a penalty should be modeled.

The penalty does not reflect the real overhead in execution cycles. Rather, it models a ranking among different jump scenarios. The lower the rank, the more favorable the situation. The point to restrain from keeping this cycle accurate is simply the overall complexity. For example, machine state and branch prediction can only be considered with unreasonably large efforts in general. A second point is that the overall model loses much of its genericity if coined for a single specific architecture. Nevertheless the actual jump penalties expressed as integral values are chosen to be worst-case values obtained by experiments on a TriCore1 architecture. An implicit consequence is that the WCET cannot be tightly obtained from the top node's WCET variable anymore. However, this value has no sigificance to the optimization.



(a) Implicit (b) Uncritical (c) Critical

Figure 4.14: Jump-penalty classification

According to the jump scenarios illustrated in figure 4.13, we define three different ranks for penalizing jump situations:

- **Implict**: Control flow between basic blocks without any explict jump (4.14a).
- **Uncritical**: Control flow between basic blocks but within the same memory (4.14b).
- **Critical**: Control flow between basic blocks in different memories (4.14c).

These ranks are assigned integral values so that the order

$$Implicit < Uncritical < Critical \tag{4.10}$$

is kept. Note that these jump penalties do not correspond to jump scenarios. Three examples for a classification are given in figure 4.14. What is achieved by such a ranking is that implicit jumping causes the smallest increase of the overall costs if applied in the ILP. This encompasses situations where blocks are connected by an implicit jump. In turn, not favoring an implicit jump means that an implicit jump in the original program will be replaced by an explicit jump when the two connected blocks are distributed in different memories. The difference of ranks *implicit* to *critical* is largest. The same applies to *uncritical* jumps. They are already composed of explicit jump instructions. Therefore, they cause a greater overhead than an implicit jump. The *critical* rank imposes the highest penalty. Therefore, the lower ranks are favored if not the gain achievable by allocating a single basic block justifies

the penalty. The concrete architecture-dependend aspects of this will be discussed below. For the *implicit* jump penalty, we chose the constant value to be 0. An *uncritical* jump is penalized with 8 cyclesThis is a safe overestimation in any case given that the exact memory timings are unknown. *Critical* jumps are penalized with 16 cycles. These values need not be tight because it their sole purpose so to group basic blocks. In practice, deviating values generally had no influence on the outcome because loops impose the largest costs in the control flow and loop bodies are therefore considered in favor of all other control flows.

The classification of jumps in this way has the consequence that now larger groups of blocks are being favored when it comes to allocation. First of all, blocks that are directly connected in the control flow graph are now considered connected within the ILP. Imposing a penalty already causes the solution to have an effect on consecutive blocks. However, jumps have different properties and especially inexpensive jump scenarios shall be kept. The ranking triggers this behavior. In contrast to *traces*, a group of basic blocks is not arbitrarily selected by only considering out- and in-degree of different control flow graph nodes but depend primarily on the kind of their context and their WCET at the same time.

An important point regarding correct jump-penalization is the order of basic blocks in the address space. For this static optimization problem, it is guaranteed that a strict partial order is kept even when blocks are allocated into the scratchpad memory.

Definition 4.4.4. A partial order is a relation "$\leq$" on a set S with the properties:

- *reflexive*: $a \leq a \qquad \forall a \in S$
- *transitive*: $a \leq b \wedge b \leq c \Rightarrow a \leq c \qquad \forall a,b,c \in S$
- *antisymmetric*: $a \leq b \wedge b \leq a \Rightarrow a = b \qquad \forall a,b \in S$

In our case, if a particular order was given in the original program then it remains to hold true for both the two partitions of blocks that will be allocated to the memories. This can easily be achieved, as the actual block placement is performed statically. The consequence is that forward jumps remain jumps in that direction within the same memory and allocation decisions have a minimized impact on the requirements to alter the instruction code afterwards. This is illustrated in figure 4.15.

The figure shows a possible outcome of an optimization. As can be seen, the assignment of blocks to different memories does not change the actual order of exectution for basic blocks per memory. It should be noted that the figure only illustrates the issue of ordering. The blocks are placed adjacent to each other in fact. The jumps between the basic blocks denoted by the arrows could be classified as *critical* and *uncritical* as defined above.

The *uncritical* jumps keep their jump direction. That is, jumps in direction of the control flow will remain. The jump between blocks b_j and b_l in figure 4.15 prevails. In fact, a possible jump at this point might become obsolete. Such situations demand for removal of such an obsolete jump and a dynamic reclassification as *implicit*. This is also something that requires consideration.

Up to this point, the fundamentals for the jump penalization have been thoroughly presented. What remains is how these ideas can be included in the overall ILP model. Clearly, the

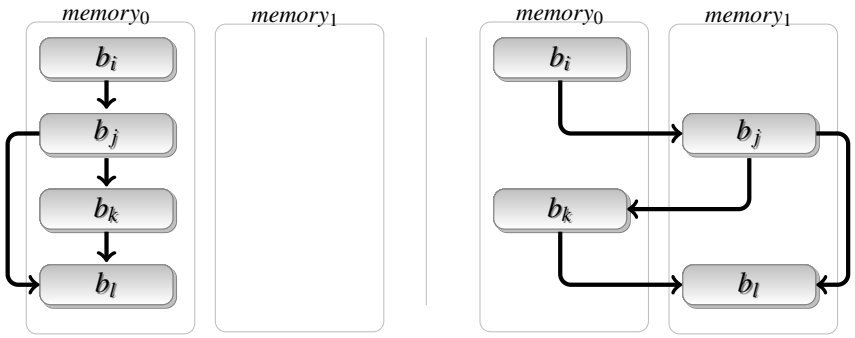


Figure 4.15: Allocation and order of blocks

classifcation of jumps in *implicit* and *uncritical* ones can already be done at compile time. But these classifications must be allowed to dynamically change, depending on the decision variables. Depending on this classification it should then be possible to let appropriate penalty values influence the decisions of the ILP solver.

Jump penalization

What can be observed is that per jump only two binary decision variables determine the classification of the jump[5]. If the two decision variables hold different values, this indicates that the two corresponding basic blocks have been assigned to different memories. This means that a jump between the two would be classified *critical*. This can be modeled by treating the two variables as operands to the logical operation *XOR* (denoted as $\otimes$). We will now investigate how to forumlate the specific jump classifications in logic operations.

An *implicit* jump can encompass at most four cases, as the two basic blocks may be assigned to two different memories. Given they are placed in the same memory together, the jump shall be classified as *implicit*. Otherwise as *critical*. Let $P_C \in \mathbb{N}$ represent the penalty of a critical jump according to the ranking defined in equation 4.10. Given two control flow graph nodes $v_i, v_j \in V$ (representing the basic blocks b_i, b_j as in figure 4.14a) and their corresponding decision varibles a_i, a_j, the jump penalty for an *implicit* jump case is defined as:

$$J^i_{Implicit} = (a_i \otimes a_j) \times P_C \tag{4.11}$$

For *unconditional* jumps the scenario is more complex. Two basic blocks b_i and b_j are connected by an explicit jump as shown in figure 4.13b. Usually, but not necessarily, such a jump bypasses a number of other basic blocks. These other blocks are not related to the current scenario because they are not part of the path in the control flow graph between the two basic blocks. However, they require consideration because they determine if a jump is required at all. As can be observed in figure 4.15, when the allocation takes place, blocks b_j and b_l become adjacent blocks in the address space of $memory_1$. Therefore, we assume a

[5] A *conditional* jump is practically treated as two separate jumps. An *implicit* and an *explicit* jump.

critical jump if the two blocks have been placed in different memories. If they have not been placed in different memories, they are at least connected by an *uncritical* jump. Such a jump is penalized by the constant $p_U \in \mathbb{N}$. In the special case of being adjacent in address space because no other blocks have been placed between the two, the additional costs of p_U need not be applied. Since the partial order of blocks is a guarantee, the set of possible blocks that could be placed between b_i and b_j can be easily obtained. It is the set of blocks that is already located between them in the unaltered program. Thus, even the *uncritical* jump penalty only applies if any of the binary decision variables $(a_{i+1}, \ldots, a_{j-1})$ corresponding to those blocks have a different value than both the variables a_i and a_j. Formally, the function determining the jump penalty in case of an *unconditional* jump for a control flow graph node v_i representing b_i is defined as:

$$\begin{aligned} J^i_{Unconditional} = &(a_i \otimes a_j)P_C \\ &+ (1 - a_i \otimes a_j)P_U \\ &- ((a_i \otimes a_{i+1}) \ldots (a_i \otimes a_{j-1}))P_U \end{aligned} \tag{4.12}$$

For *conditional* jumps, the two succeeding basic blocks b_j, b_k need to be considered. This is the situation in figure 4.13c. To properly penalize such a jump, all combinations of the three binary variables corresponding to the basic blocks need consideration. On the one hand, there exists the implicit jump from b_i to b_j. It corresponds to the simple scenario expressed in equation (4.11). Either, both are placed in the same memory, which means the jump is *implicit*. Otherwise both are located in two different memories, then the jump is penalized as a *critical* jump. Similarly, the explicit jump penalty is *uncritical* as long as both basic blocks b_i and b_k remain in the same memory. It is *critical* otherwise. Just as in the *conditional* case, when there are no basic blocks located in the address space between the two basic blocks, the penalty need not be applied. This is because the explicit jump would be unnecessary. Let a_i, a_j and a_k correspond to the respective basic block assignments and v_i to the leading basic block b_i, then the penalty for a *conditional* jump case is defined as:

$$\begin{aligned} J^i_{Conditional} = &(a_i \otimes a_j)P_C \\ &+ (a_i \otimes a_k)P_C \\ &+ (1 - a_i \otimes a_k)P_U \\ &- ((a_i \otimes a_{i+1}) \ldots (a_i \otimes a_{j-1}))P_U \end{aligned} \tag{4.13}$$

The penalties that are applied to the constraints in the ILP model is denoted by the function c_{jump} defined as:

$$c_{jump} = \begin{cases} J^i_{Implicit} & \text{if the jump case is } \textit{implicit} \\ J^i_{Unconditional} & \text{if the jump case is } \textit{unconditional} \\ J^i_{Conditional} & \text{if the jump case is } \textit{conditional} \end{cases} \tag{4.14}$$

The different jump scenarios and the necessary classification can be detected at compile time. Placing different basic blocks into another, faster memory results in a gain as defined earlier. This gain comes with a cost. The cost is determined by the instruction code that will

require modifications to construct a valid program again, after the placement of blocks. The fact that direct connections of basic blocks have now also been modeled, these connections will now also be taken into consideration when searching for a solution. This also leads to a grouping of basic blocks, since separating a basic block from a sequence of blocks will now imply an overhead. Given such a decision is made within a loop, this overhead is even multiplied by the number of loop iterations. Therefore, the jump penalty becomes a considerable factor in finding an overall solution.

What remains is the problem of encoding the logical operations in a form that is a valid input to an ILP solver. The required format is that of a set of (in)equations. These must be modeled so that we achieve correct results given the overall objective of minimization. The logical function *XOR* can be expanded by defining it in terms of:

$$a \otimes b = (a \wedge \overline{b}) \vee (\overline{a} \wedge b) \tag{4.15}$$

The logical *AND* function can be expressed as a set of inequations in the following way, given that all variables are already restricted to the interval [0,1]:

$$x = a \wedge b \quad \rightarrow \quad \begin{array}{rcl} x & \geq & a+b-1 \\ x & \leq & a \\ x & \leq & b \end{array} \tag{4.16}$$

Under the same restriction, the logical *OR* function can be expressed in terms of:

$$x = a \vee b \quad \rightarrow \quad \begin{array}{rcl} x & \leq & a+b \\ x & \geq & a \\ x & \geq & b \end{array} \tag{4.17}$$

Since all variables are binary the logical function *NOT* is trivially expressed as:

$$x = \overline{a} \quad \rightarrow \quad x \;=\; 1-a \tag{4.18}$$

With this basic knowledge at hand, we can compose the equations that make up a logical *XOR* operation on binary operands:

$$\begin{array}{rcl} x & = & (a \wedge \overline{b}) \vee (\overline{a} \wedge b) \\ & = & x_1 \vee x_2 \end{array} \quad \rightarrow \quad \begin{array}{rcl} x & \leq & x_1 + x_2 \\ x & \geq & x_1 \\ x & \geq & x_2 \\ x_1 & \geq & a + (1-b) - 1 = a - b \\ x_1 & \leq & a \\ x_1 & \leq & 1 - b \\ x_2 & \geq & (1-a) + b - 1 = -a + b \\ x_2 & \leq & 1 - a \\ x_2 & \leq & b \end{array} \tag{4.19}$$

Another problem we need to solve is the chaining of logical *XOR* functions as (in)equations 4.12 and 4.13. There, all the single results of the *XOR* subterms form a chain of conjunctions.

As a whole, the term evaluates to the value 1 if and only if all the single functions evaluate to 1. The second observation is that the left-hand side operand a_i is the same throughout all subterms. From these facts we can obtain a single set of expressions that covers the conjunction of logical *XOR* functions under the restriction that one and the same operand appears in each subterm. We can transform the original term so that:

$$\begin{aligned}
&(a_i \otimes a_{i+1}) \wedge \ldots (a_i \otimes a_{j-1}) \\
&= ((a_i \vee \overline{a_{i+1}}) \wedge (\overline{a_i} \vee a_{i+1})) \wedge \ldots ((a_i \vee \overline{a_{j-1}}) \wedge (\overline{a_i} \vee a_{j-1})) \\
&= (a_i \wedge \overline{a_{i+1}} \wedge \ldots \overline{a_{j-1}}) \vee (\overline{a_i} \wedge a_{i+1} \wedge \ldots a_{j-1})
\end{aligned} \tag{4.20}$$

This form is already similar to the one in equation 4.15. The transformation into a set of inequations is also very similar. Given that now $|\{a_{i+1}, \ldots, a_{j-1}\}| = j - i - 2 = n$ operands are applied with a_i instead of just one, equation 4.20 resolves to:

$$\begin{aligned}
x &= (a_i \wedge \overline{a_{i+1}} \wedge \ldots \overline{a_{j-1}}) \\
&\quad \vee (\overline{a_i} \wedge a_{i+1} \wedge \ldots a_{j-1}) \\
&= x_1 \vee x_2
\end{aligned}
\quad \rightarrow \quad
\begin{aligned}
x &\leq x_1 + x_2 \\
x &\geq x_1 \\
x &\geq x_2 \\
x_1 &\geq a_i - \textstyle\sum_{k=1}^{n} a_{i+k} \\
x_1 &\leq a_i \\
x_1 &\leq 1 - a_{i+1} \\
&\vdots \\
x_1 &\leq 1 - a_{j-1} \\
x_2 &\geq -a_i + \textstyle\sum_{k=1}^{n} a_{i+k} - (n-1) \\
x_2 &\leq 1 - a_i \\
x_2 &\leq a_{i+1} \\
&\vdots \\
x_2 &\leq a_{j-1}
\end{aligned} \tag{4.21}$$

Putting together the insights we obtained and the templates for equations for some fundamental logic operations, we can now extend the very basic control flow constraints discussed in section 4.4.1. Specifically, the constraints defined in equations 4.1, 4.2 and 4.3. This is simply done by extending the constraints by the jump pentalty function c_{jump} defined above. Therefore, the general form of a control flow dependency constraint - given the WCET variables $\{w_i \mid \text{WCET variable of node } v_i\}$ and the cost function c - is now:

$$w_i \geq w_{i+1} + c_{main}(v_i) - gain(v_i) \times a_i + c_{jump}(v_i) \tag{4.22}$$

Similarly, this is applied to branch and loop constraints.

Summing this up, we have now fully assembled the components to a complete ILP to model the control flow graph of a function. We solved the problem of discovering the WCEP by only implicitly modeling the path through control flow constraints. This allows for an efficient processing on the one hand and on the other it allows to extend these constraints to achieve specific optimizations goals along the WCEP. Also, the problem of grouping basic blocks, if this is beneficial, has been solved. No static preselection is performed. By imposing jump penalties we have a dynamic control over whether blocks become separated depending on the balance of gain and penalty. Under this model, we can obtain optimal results for the static, WCET-centric code allocation problem for single functions.

Global control flow

Next, this model needs to be extended so that complete programs can be covered. For this purpose, we need to discuss some aspects of program-wide control flow prior to extending the system of equations.

A program consists of a set of functions. Currently, the ILP only models the intraprocedural control flow of a program. One of those functions is the dedicated entry point of the program. This is illustrated in 4.16.

Calling function f_1 from f_0 in general implies that the control flow will return to the point in f_0 from which the call has been performed. Or more specifically, to the instruction following the call. Because of this, the global entry function is also the global exit function.

If we were to establish an interprocedural control flow graph that models the invocations among the functions, a unique *source* node and a unique *sink* node could be identified. Depending on the system and the programming language, this structure need not necessarily exist. In the C programming language the control flow can be altered in irregular ways. Namely, the function pair *setjmp()/longjmp()* allows to create a backup of the current context on the call stack and to restore it. This means that the order of function invocations does not necessarily match the reverse order of function returns. Another issue is the use of the *exit()* function which allows immediate termination of the program. These two situations, however, rarely arise. What is required to extend the ILP for interprocedural control flow, is that every path in the program's control flow graph spawns at the source and ends at the sink of the dedicated entry function as shown in figure 4.16 (with f_0 as such a function).

This restriction reveals yet another problem. In the C programming language, not all functions contributing to the global control flow need to be known at compile time. As long as a function is declared, it need not be defined. Since the optimization we are performing is integral part of the compilation process, all functions involved must be known in their respective intermediate representation. Otherwise, it is not possible to reason about the WCET. In fact, this is not a limitation of this optimization but a vital requirement for all WCET-related processing.

A call is not treated as an explicit jump as we investigate a function's control flow. Still, a call terminates a basic block. This implies that such a basic block performs an implicit jump to its successor. The local jump penalty is established accordingly. We will refer to such a basic block as a *caller*. The control flow will continue within the function at the call block's successor when the invoked function returns. The situation is also illustrated in figure 4.16. The pair of nodes in a function's control flow graph that are involved in calling and returning is called a *call site*.

Given these restrictions, the observation can be made that if function f_0 invokes another function f_1, the WCET of the former rises by exactly the WCET of the latter. The WCET of f_1 can be obtained through the WCET variable corresponding to the source node of f_1's intraprocedural control flow.

A function can be called from multiple points. The solution to modeling this calling is to introduce a function that maps the entry node of the callee to its WCET value. That is, the

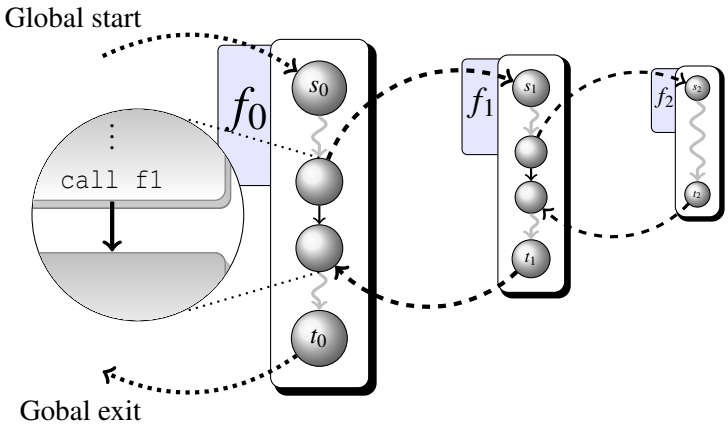


Figure 4.16: Handled program structure

per function WCET. This is simply possible as functions may only have a single entry point. To represent that cost for all possible call sites, all the ILP constraints need to be extended. This is done in the same way as it was done for the jump penalties. Therefore, let f_n denote the current function and f_m a possibly called function, then

$$e^{f_n} : V^{f_n} \rightarrow \mathbb{N}_0$$

represents the total WCET of the callee, identified by the calling node of the call site as defined up to this point. It is understood that such call costs only apply on nodes that actually perform a call. We define a separate jump penalty μ for calls as:

$$\mu : V^{f_n} \times V^{f_m} \rightarrow \mathbb{N}_0$$

with

$$\mu(v_i, v_j) = (a_i \otimes a_j) p_C + (1 - a_i \otimes a_j) p_U . \tag{4.23}$$

Where a_i and a_j correspond to their control flow graph vertices in either function. This function will either penalize a call as critical jump if both the operands have been allocated to different memories. Otherwise the penalty is that of an uncritical jump.

The function $c^{f_n}_{call}$ itself is defined as

$$c^{f_n}_{call}(v_i^{f_n}) = \begin{cases} w_0^{f_m} + \mu(v_i^{f_n}, v_0^{f_m}) & \text{if } f_m \text{ is called} \\ 0 & \text{otherwise} \end{cases} \tag{4.24}$$

Where $w_0^{f_m}$ is the WCET variable of the entry node in function f_m.

The general form of the extended ILP dependency constraints is now defined as:

$$w_i \geq w_{i+1} + c_{main}(v_i) - gain(v_i) \times a_i + c_{jump}(v_i) + c_{call}(v_i) \tag{4.25}$$

The extension equally applies to branch and loop constraints. It is the finalized constraint which is used in the ILP model.

At this point a program is fully described. The costs that each function contributes to the overall WCET is simply added to the nodes that represent the calling blocks. This also simply covers the fact that one and the same function can be invoked multiple times from different locations. The total costs accumulate accordingly. Explicit information on the control flow is not necessary. Representing that by an interprocedural control flow graph would reveal the order in which functions are being invoked. But this is not necessary at this point, thus keeps the construction for the ILP for the static allocation problem simple.

Dynamic growth

There still exists a major problem we have not addressed yet. Above, it was emphasized that allocating basic blocks into a different memory usually requires a modification of the instruction code. Due to the nature of RISC instruction sets, only limited possibilities exist to bridge large intervals in address space when it comes to jumps or calls. The model so far uses a static bound and assumes static sizes per basic block. That means that the space available is consumed up to its maximum. The problem is that the ILP decisions - as a result under this size constraint - will dictate an allocation that inevitably leads to a growth of size due to code modifications. Unless the size bound was zero, of course. A way must be found to address this problem dynamically within in the ILP. The goal should be to obey the size bound strictly.

Fortunately, the fundamentals to get hold of this are already available. The jump penalization already provides a way to deal with different jump scenarios. Code modifications are only required for jumps and all possible jumps have already been identified. For jump penalization an overhead was added to the overall WCET, depending on the jump scenario. We can easily reuse this technique to modify the global size constraint to depend dynamically on those cases. Because it is not possible to exactly model the required code modifications, the worst case is constantly assumed. This means, the largest instructions that possibly could be emitted are assumed. In turn, this means that the scratchpad memory is not necessarily completely filled.

At this point, only the required extensions to the overall model will be presented. The actual correction of jumps is a complex task due to the nature of the TriCore1 instruction set and will be discussed in detail separately below. Therefore, the following discussion is kept at an abstract level to avoid repetition.

The following constants represent the worst-case size overhead a jump scenario can cause:

- E_I: An implicit jump initially consists of no instructions altering the control flow. The worst-case scenario is the placement of its successor into a different memory. An explicit unconditional jump instruction is introduced.
- E_U: An existing explicit unconditional jump might not be capable of bridging the large distance through the address space and need to be replaced accordingly.
- E_C: In the case of a conditional jump, two successors can be jumped to. One jump is implicit, the other is explicit and conditional. In the worst case, both the successors

are placed into different memories. The conditional jump in general cannot bridge large address space intervals and needs to be replaced by an indirect jump towards the target block. Moreover, the implicit jump needs to be replaced by an appropriate explicit jump.

- E_F: This constant covers the case of a basic block that terminates with a call. If the target is located in another memory, the call might need to be replaced by an indirect call which allows for arbitrary targets.

Assembling these worst-case estimations, a function similar to the one defined for jump penalization can be defined. It will be applied to the global size restriction.

Let s_{dyn} denote the function that performs a worst-case estimation on the dynamic growth of a single basic block b_i, respectively its corresponding control flow graph node v_i. The possible successors of v_i are v_j and v_k. The function is defined as:

$$s_{dyn} : V \rightarrow \mathbb{N} \tag{4.26}$$

with

$$s_{dyn} = \begin{cases} (a_i \wedge \overline{a_j})E_I & \text{if the jump is implicit} \\ (a_i \wedge \overline{a_j})E_U & \text{if the jump is unconditional} \\ (a_i \wedge \overline{a_j})E_I + (a_i \wedge \overline{a_k})E_U & \text{if the jump is conditional} \\ (a_i \wedge \overline{a_j})E_F & \text{if the jump is a call} \end{cases} \tag{4.27}$$

Given this estimating function, the static global size restriction is defined as

$$\sum_{v_i \in V} (s(v_i) + s_{dyn}(v_i)) \times a_i \leq size_s \tag{4.28}$$

Objective function

The final issue to address is the ILP objective. As was already mentioned, the objective is to minimize the WCET variable of the entry node of the program. Let $v_0^{main} \in V^{main}$ denote this node, then the objective of the overall ILP model is:

$$v_0^{main} \rightarrow min \tag{4.29}$$

The model for the static allocation problem is now complete. It was shown how an ILP model can be constructed that performs static allocation decisions. The novelty is that the allocation decisions are WCET-centric. While other existing approaches required explicit path enumeration, a possible way was presented to make use of implicit path information. The primary advantage is that a solution can be obtained by solving a single problem once. In addition, technical problems concerning code modifications were addressed. Interestingly, this point has often been neglected in related works - irrespective the optmization goal (WCET, energy, etc.). It was shown how this problem can be handled by estimating the dynamic growth within the ILP.

4.5 Implementation

In this section, various aspects of the actual implementation are presented. Primarily, the generation of the ILP model and the modification of the source program due to the ILP decisions are discussed. The latter is highly architecture-dependend. Although some of the problems that arise are specific to this architecture, for the most part the presented techniques apply to many other RISC in general. The workflow of the complete optimization is shown in figure 4.17.

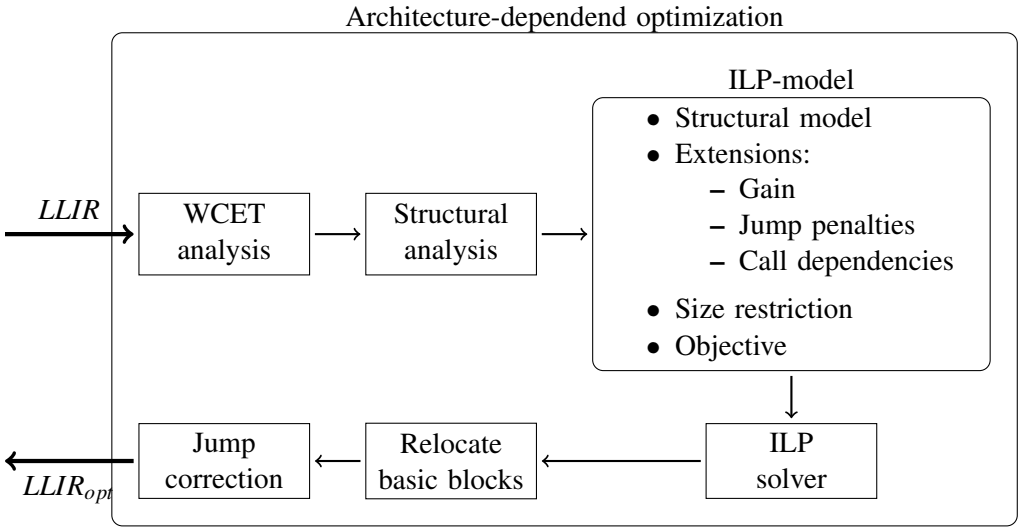


Figure 4.17: Workflow of static optimization

4.5.1 ILP construction

The construction of the ILP model fundamentally depends on the results of the structural analyis presented in section 4.2.2. This analysis leads to the creation of a graph that represents the program structure in an hierarchical manner. This graph is referred to as a *control tree*. An example of such a data structure was given in figure 4.5b. As can be seen, the tree allows to identify nesting of high-level constructs of the programming language. In the light of the discussion of the ILP model above, such a data structure proves useful in two respects. On the one hand, an ILP model can be obtained from the control tree almost directly. The type and the nesting of program structures is well described, therefore the creation can be performed by simply traversing the tree. On the other hand, the extensive information the control tree offers proves helpful when it comes to handling exceptional code structures. An example of this are irregular exits from loops as discussed in section 4.2.2. If we were only to examine the structure of the control flow graph on a per-node basis those exceptional cases would need to be identified separately. As the name suggests, a control flow graph only presents flow information. Structural information is limited insofar that only node-degrees and adjacency can be easily obtained. The use of the control tree cleanly separates the steps

of identifying structure and the generation of the ILP.

As was just mentioned, an ILP can be generated from a given program by traversing the control tree. The source program comes in the LLIR representation (see section 3.2.4). As was said, the LLIR represents the physical layout of the program as well as the actual control flow. For our purpose, it is convenient to extract the control flow information from the LLIR and generate a set of control flow graphs explicitly. The reason for this is that these graphs serve as input into the structural analysis. While generating the control tree, the control flow graphs will be reduced by replacing identified structures by single nodes. This repeats until the whole control flow graph has been reduced to a single node. This can more easily be done with a custom, simplified graph than operating on the LLIR directly.

The control flow graphs are generated on a per-function basis. This is due to the fact that the control trees and the generation of the ILP rely only on the information of a single function. This implies that information on the full program is not available at any point during the creation of these data structures.

However, while traversing the LLIR to create the control flow graphs a map of function calls is created as well. This allows for a validity check afterwards. The input is valid if all functions possibly called at runtime are available as input into the optimization.

After the structural analysis succeeds, the control tree can be traversed in post-order. For each of its nodes, the necessary constraints can be created. At this point, it is advantageous that loops have already been classified. Therefore, it is easier to create the loop constraints as described in 4.4.2. Also, constructs that need no explicit representation in the ILP are already known and can be omitted (like *break* statements).

The ILP requires that information on the size and the WCET of single basic blocks and the iteration counts of loops are statically available. This information gets into the ILP as constants. Therefore, they need to be obtained previously. The size of objects can be trivially obtained from the LLIR. The per-block WCET is not available right away. Moreover, the ILP depends on the WCET to be known for both the execution in the main memory as well as in the scratchpad memory. Because of this, we perform two WCET analysis steps on the whole program. As was described in section 3.4, the LLIR was extended to feature capabilities of a linker. This allows to model the physical setup of a program before an object file is actually created. We use this ability here to mark the program as being loaded into one of the two memories. For each assignment, the integrated WCET analysis is invoked. After each step, the blockwise WCET is extracted from the LLIR and stored for later use. The iteration count for loop constructs is assumed to be annotated in the source code or determined by an explicit loop bound analysis. The optimization expects this information to be readily available and accessible through the LLIR. We are only interested in the upper loop bounds.

The traversal of the control tree leads to an ILP we will refer to as the *structural model*. It contains information on control flow dependencies only. The actual terms that model the problem will be added later. The primary reason for this multi-step approach is that the structural model should be kept reuseable for other, differently aimed optimizations as well. Specifically, the dynamic allocation problem presented in chapter 5 relies on the structural model to be available.

In a second step, the structural model is extended so as to express the specific problem we are trying to solve. For this purpose, the terms for expressing the WCET gain (eq. (4.6)) and jump penalties (eq. (4.11), (4.12), (4.13)) are added to the existing constraints. For basic blocks containing calls, additional jump costs are added which model the expense of a call to another function. After that, the size restriction according to eq. (4.28) is set up.

The final step is to define the overall objective of the ILP. It is to minimize the overall WCET. Therefore, the value of the WCET variable that represents the entry node of the program is to be minimized.

When the ILP is solved, the binary decisions concerning the placement of basic blocks are available. To actually perform the relocation, use is made of the extension to the LLIR that allow the assignment of LLIR objects to different object sections in the output file. The principles have been throughoutly discussed in section 3.4.

4.5.2 Program correction

The relocation of basic blocks into a different memory by means of object section assignments does not automatically imply that the result is a working program. Although the linker will handle symbolic references according to the defined program layout, the instructions themselves will no be altered. This is a major problem on many RISC architectures. The code selection phase of the compiler seeks for optimal instruction generation. Optimal instructions are the most efficient ones regarding their memory footprint.

On an architecture like the TriCore1, instruction widths of 16bit and 32bit exist. For 16bit instructions, at most 8bits are devoted to the opcode which severly limits the size and the number of explicit operands. This limitation becomes most apparent when it comes to immediate values as operands. The solution is usually to indirectly access a value through a previously loaded temporary register. This is clearly wasteful and therefore the code selector in general picks the smallest instructions with preference. In general, this is no problem. Either data is dynamic, then it is inevitable that a register is used. The data could either stem from a register of a memory location. Or the data is statically known, such as constants from arithmetic expressions. In this case the appropriate instructions can already be selected. When it comes to altering the control flow through jump or call instructions, matters are different however. Although the jump targets are known by their symbolic names, during code selection it is not always possible to determine the distance to the target. This has two reasons: On the one hand, the jump could be a local one. In this case the sequence of instructions can still change since the code selection is not done yet. Therefore, offsets to targets cannot be determined in a single pass. The second reason is that the availability of jump targets is no requirement at all. The correction of unresolved symbols is up to the linking phase. As our requirement is that the complete program is available, only the former issue is of concern.

Because of the reasons given, the strategy of the WCC is to emit the smallest jump instructions possible on code selection and to later adjust all instructions with unsuitable displacements. This is done by summing up the instruction sizes between two symbolic labels to

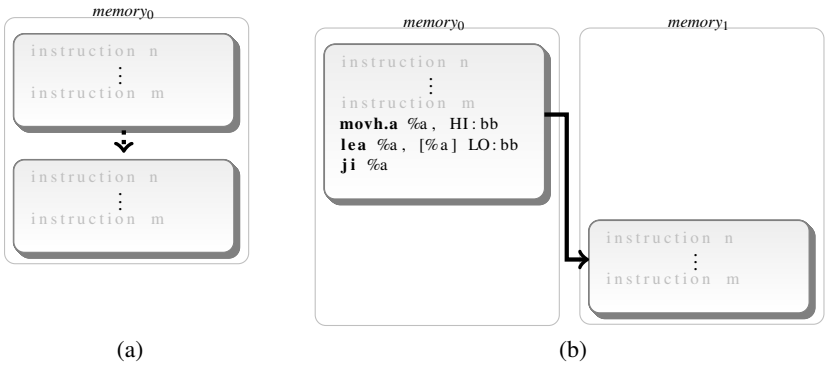


Figure 4.18: Implicit jump correction

determine the distances and to replace the existing instruction. Such a replacement can inflate the size of a given basic block. This can be the replacement of a 16bit by a 32bit jump instruction so that a greater width for an immediate operand is provided. If no suitable single instruction is available, the target needs to be loaded into a separate register and an indirect jump is performed. Apart from a bigger memory footprint the allocation of a register is no problem at this point of the compilation as the register allocation has not been performed yet, therefore registers are still virtual.

On TriCore1, there exist 31 different jump instructions which themselves partly come in 16bit and 32bit variants. The width of the immediate operands ranges from 4bit on 16bit conditional jump instructions to 24bit for the unconditional jump. This is sufficient to provide suitable instructions for all adjustments within a single function. There exist 16 memory banks. Each of which has a width of 16MB. In our case, a jump correction also has to be performed. However, now the jump distances are not limited to a single memory bank. It must be possible to cover the full address space. The memory banks on Tricore1 are placed in intervals of 256MB each. If we are to move a basic block from one memory into another, the existing displacement correction cannot be applied. The fact that the jump targets can now be significantly more distant, increases the basic block inflation even more. For displacments that far, it is inevitable to perform indirect jumps. This is not always a matter of simple instruction replacement restricted to a single basic block.

We have briefly investigated the issue of dynamic growth of basic blocks in section 4.4.2. There, we defined constants that reflect the worst-case inflation when it comes to jump correction. In the following we will investigate the cases in detail.

Figure 4.18 outlines the correction of an implicit jump. Instead of executing the next instruction when exiting the leading basic block, an explicit jump is introduced. Unfortunately, there exists no single jump instruction on TriCore1 to bridge the given jump distance. Since it is not even possible to load a jump target directly due to the described limitations of operands, a pair of instructions loads the target address. After that, an indirect jump can be performed. This is outlined in 4.18b

In the case of an unconditional jump (figure 4.19), the exiting jump instruction need to be replaced by the same sequence of instructions as in the implicit case.

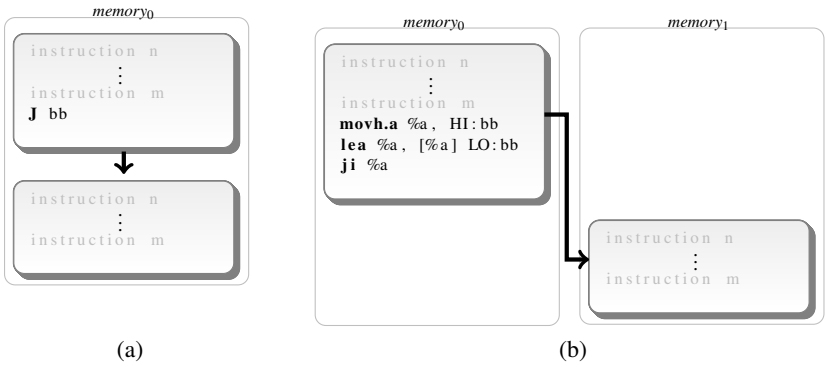


Figure 4.19: Unconditional jump correction

For conditional jumps as outlined in figure (figure 4.20), different cases require consideration. There exist two target blocks. Each of which could be placed into another memory. If the implicit target is placed into another memory than the leading block, as shown in figure 4.20b, then an aritificial basic block needs to be created, since a jump by definition terminates a basic block. This *trampoline* block then indirectly jumps to the effective target similarly to the cases already described. A similar correction is required when the explicit target of the conditional jump is relocated (figure 4.20c). The target in the local memory is replaced by a trampoline block which in turn jumps to the real target. If both target blocks are relocated two trampolines need to be generated as shown in figure 4.20d.

The changes to the instruction code just outlined are applied to every single basic block. These changes are symmetric regarding the placement of basic blocks in main memory and scratchpad memory. All of them lead to increasing sizes of basic blocks. The figures only demonstrate the most basic replacement strategies. In fact, some optimizations are possible. Given the case in figure 4.20c, by reversing the jump condition, the explicit target block would become the implicit one and an explicit jump could possibly be saved. Also, it is often possible to use 16bit instructions instead of their 32bit variants as jumps within the same memory for the presented scenarios are often small. In the estimations for basic block inflation, only the worst cases are considered. This is because modeling the block distances in the ILP would greatly increase its complexity. Because of this overestimation, small amounts of space in the scratchpad memory can possibly be wasted. The instruction adjustment implemented for this thesis strives for optimal replacments, making constantly use of the most appropriate modifications taking the full scale of jump instructions into consideration and optimizes by altering jump conditions or the removal of obsolete jumps.

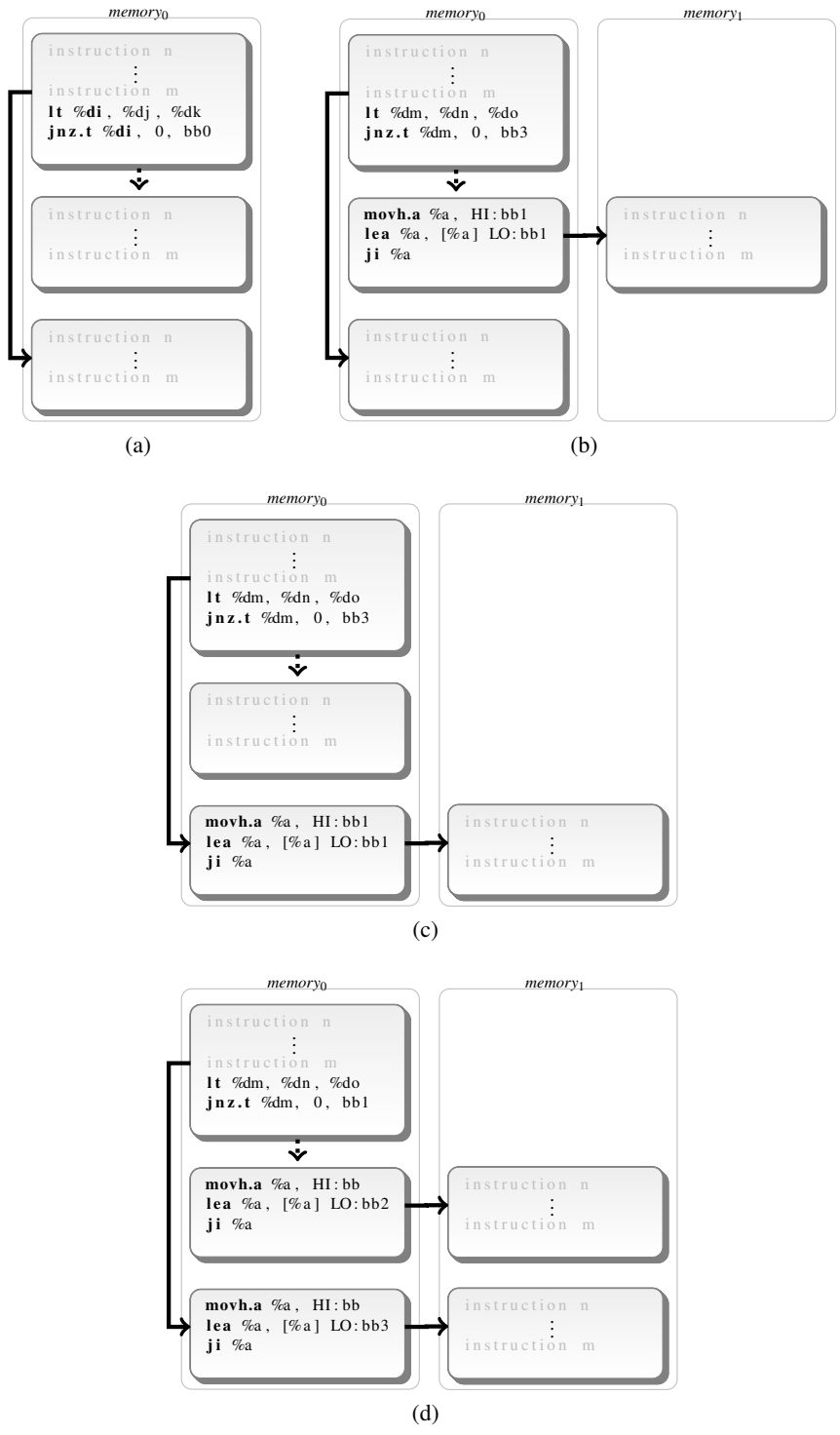


Figure 4.20: Conditional jump correction

CHAPTER 5

DYNAMIC SCRATCHPAD ALLOCATION

In this chapter, a strategy for dynamic allocation of instruction code is presented. As opposed to the static allocation presented in chapter 4, the contents of the scratchpad memory are not fixed in this case. During execution, only parts of the program are copied into the faster memory to achieve an improvement. This is an advantage over the previous approach in the respect that programs too large to fit entirely into the scratchpad memory are now allowed to be partially allocated only. Moreover, no memory is wasted by keeping unused code.

The dynamic allocation is fully determined at compile time. This is similar to the static allocation where code fragments are preselected and placed. In this case, the allocation decisions must not only take into account the bare WCET counts of the objects but also the execution history. In particular, the decisions must be based on the knowledge of when certain fragments are executed and when they are never executed again during a program's lifetime.

In section 5.1, a survey is conducted on the related works in the field of dynamic allocation strategies. This will reveal the problems that are related to such techniques and motivate the strategy we are to follow within the rest of the chapter.

The allocation decisions largely depend on information about the dynamic behavior of a program. This topic is handled in section 5.2. A basic method for the determination of object lifetimes is presented. However, this is applied to isolated functions only, which makes it unsuitable for our particular problem since the aim is to optimize complete programs. Therefore, a variant is developed which is specifically aimed at determining lifetimes globally.

The allocation problem is solved by utilizing an ILP model, just as in the static allocation approach. In section 5.4, this ILP model is formally constructed. The presented approach derives ideas from related works, but as a whole forms a unique approach towards solving the WCET-directed dynamic allocation problem for code.

Section 5.5 deals with the postprocessing of the ILP results. As we will motivate in section 5.3, solving the problem is performed in two successive stages to reduce the complexity of the ILP.

A key problem in the dynamic allocation is the ability for self-modifications by transferring objects between memories during execution time. This is achieved by supplementing the original program with additional code, which is discussed in section 5.6.

Following this, the technical realization is discussed in section 5.7. After the allocation decisions have been determined, the program requires modification to enable the dynamic copying of its own instructions into a different memory. Also, problems concerning the WCET analysis and the dynamic program behavior are handled.

5.1 Related work

Some of the relevant work for this chaper related to the allocation to a scratchpad memory has already been presented in chapter 4. Similar to the static allocation schemes, the first approaches towards dynamic allocation were related to using caches. A cache has the advantage that its control is transparent to the running process. Thus, optimizations that make use of them are not necessarily required to make any explicit changes to the program code to achieve a distribution among memories. As was presented, caches were utilized by statically loading and locking their contents. As opposed to that, dynamic techniques have been proposed that allow the contents of caches to be changed during execution [Pua06].

Data and instruction codes have different properties. Data accesses can potentially be performed from everywhere within the executing code. If small amounts of data are being accessed, a data cache would potentially store much unneeded information. Opposed to that, instructions have a high degree of locality. Therefore, filling cache lines sequentially potentially yields good results. Because of this, some embedded architecture like TriCore1 only include an instruction cache but no data cache.

In [RND+05], a dynamic instruction placement strategy is presented. Its aim is to reduce the power consumption by making use of cache memories. The problem is solved by constructing a problem which can be solved by dynamic programming techniques. They optimize for energy consumption or average execution times.

It was already mentioned that the use of caches is generally problematic because of their unpredicability. Despite their advantage concerning the technical realization, the dominant approach to dynamic allocation is to make use of the scratchpad memory. [EKJ+06] propose an optimization that attempts to determine an optimal mapping strategy for *pages* of instructions. A page is considered equivalent to a cache line. For uniformely sized objects, the memory management is significantly less complex.

Instead of deciding the allocation for fixed units, [VWM04b, SGW+02] investigated ways to achieve optimizations by preselecting objects. For data, this is restricted to single variables. For instructions however, the locality of execution is accounted to select sequences of instructions as the basic unit of allocation. The disadvantage over fixed sized units is that fragementation on the scratchpad potentially occurs.

For caches, a request for loading a cache line has to be made, whereas for scratchpad memories, an explicit copy routine has to be executed that moves the objects accordingly. The lat-

ter is similar to spill decisions in the domain of register allocation. [DP07] and [VWM04b] consider the allocation problem similar to the register allocation to a single large register, which is the scratchpad memory. Both papers propose the construction of an ILP.

The dominant approach in the papers above is that a fixed set of *reload points* (or *spill point* is determined on which the dynamic allocation is performed. These are typically the execution points before loops and at the entry to functions. Opposed to that, [VWM04b] proposes a strategy which allows their selection dynamically, depending on their actual benefit. These techniques aimed at either reducing the average execution time or the energy consumption of energy. However, the algorithm is limited to the allocation of data objects. In [PP07], a comparison of dynamic allocation techniques utilizing caches as well as scratchpad memories is made. However, their own proposed optimization technique uses a greedy approach for contents selection. All of the presented techniques precalculate the spill decisions at compile time. Therefore, although a dynamic behavior is imposed, the problem is still to be considered static. At runtime no calculation is ever performed.

From these papers, it is apparent that the problem of allocating instructions into a scratchpad memory with the aim of WCET reduction has not been addressed yet sufficiently. Although approaches exist, they are either not at all directed towards reducing the WCET or are limited by the fact that every allocation decision potentially requires a reevaluation of the WCET timings. Thus, only a greedy approach has been presented. This is the same problem that already motivated the WCET directed static allocation on the previous chapter. No comprehensive technique has been proposed yet which is capable of solving the problem of dynamic WCET-centric code allocation.

5.2 Interprocedural lifetime analysis

In this section, a technique for the determination of object live ranges in a program is presented. In 5.2.1, the necessity of such an analysis in the context of the dynamic allocation problem is motivated. In 5.2.2, a special class of control flow graphs is introduced that assist in analyzing execution behavior at a program wide scope. Afterwards, in section 5.2.3, an algorithm is presented to obtain information live ranges from the program wide control flow graph.

5.2.1 Motivation

For solving the static allocation problem, we construct control flow graphs from functions represented by the LLIR. Although the LLIR itself provides access to control flow information, the newly generated graphs are specialized to allow for an easier construction of the final ILP. In particular, the ILP was generated by considering the structure of isolated functions. Calls among the functions could easily be modeled as extensions to the basic constraints. It was sufficient to collect information on function calls by simply iterating the instructions of the LLIR and to recall which functions were invoked. The order in which these calls are performed is irrelevant in the static allocation problem. All allocation deci-

sions are fixed. Objects that have been selected for execution from within the scratchpad memory remain in their places throughout the lifetime of the program. Therefore, it is unnecessary to obtain and to manage information on the execution history.

Opposed to that, the strategy of the dynamic allocation presented in this chapter is to move objects into the scratchpad memory during execution. This implies that some information on when objects are executed must be available. In particular, the scratchpad can only be used optimally if relevant objects are moved into the scratchpad before they are executed. In addition, it must be possible to overwrite them with others if we can guarantee that they are never executed again. In general, the order of execution is central to solving the dynamic allocation problem.

As was mentioned, intraprocedural control flow graphs were sufficient for solving the static allocation problem. Now, instead of only knowing if a function is called, we need precise information on when this happens, so that a reasoning about the execution history becomes possible. For this reason an *interprocedural control flow graph* is constructed which serves this requirement.

5.2.2 Interprocedural control flow graphs

The interprocedural control flow graph (IPCFG) we are going to construct will reflect the execution history of the complete program throughout its lifetime. Equally to an intraprocdural graph, all paths through it start at a unique source node and end at a unique sink node. Every such path through the IPCFG reflects the execution order of a program's basic blocks. More formally:

Definition 5.2.1. An **interprocedural control flow graph** (IPCFG) is a control flow graph $\hat{G} = (\hat{V}, \hat{E})$ (according definition 2.2.2) whose nodes represent all basic blocks of all functions f_i, so that $\hat{V} = \bigcup_i V_i$. In addition to the edges that determine the order of execution within single functions, edges E_{call} and E_{return} model the call to a and the returning from a function. $\hat{E} \subseteq \bigcup_i E_i \cup E_{call} \cup E_{return}$.

The set of edges is not fully equivalent to the conjunction of edges of separate function. This is due to the way calls and returns are modeled, as is discussed shortly.

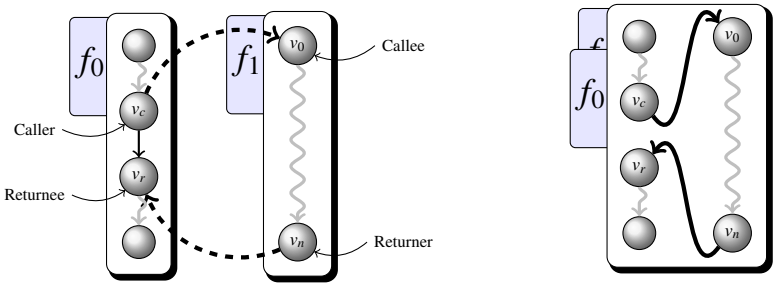
In an IPCFG, function calls have an explicit representation. For a CFG that is limited to a function, no objects outside the function are considered. Because of this, it is not possible to directly lookup the complete path of execution throughout the program. Control flow graphs limited to single functions are therefore inappropriate.

However, we can extend existing intraprocedural graphs. In chapter 4, it was discussed that all such graphs also have a unique source and sink nodes. Multiple entries were explicitly forbidden and we found that in code generated from the C programming language, functions with multiple entries aren't even possible at all. Such code may realize multiple exits from a function. This was solved by extending the corresponding control flow graph with a virtual sink node and by connecting the existing sinks with it (see figure 4.9(2)).

This structural guarantee allows for the modeling of calls between functions by means of edges between the calling basic block (*caller*) and the entry block of invoked function (*callee*). When returning from functions, the execution continues with the basic block following the caller. Figure 4.16 illustrated the situation. We will refer to this block as the *returnee*. Equally, the basic block containing a return instruction will be referred to as the *returner*.

To generate an IPCFG from a set of CFGs, nodes representing callers and returnees and their connecting edges are replaced by *call sites*.

Definition 5.2.2. A **call site** is a pair of nodes $\{v_c, v_r\} \in V_{f_i}$ in a control flow graph $G_{f_i} = (V_{f_i}, E_{f_i})$ where v_c represents the caller and v_r represents the returnee. The nodes are not directly connected but form the head and the tail of all paths $P = (v_c, v_o, \ldots, v_n, v_r)$ with $\{v_0, \ldots, v_n\} \in V_{f_j}$ from $G_{f_j} = (V_{f_j}, E_{f_j})$, which represents the called function.



(a) Original CFGs (b) Interprocedural CFG

Figure 5.1: Control flow for calls

To make this more clear, figure 5.1 illustrates the construction. Although the caller and the returnee are connected in the original CFG to denote the implicit jump, they are now indirectly connected by following any path through the called function. The transformation just described is applied to all callers of all CFGs. An implication of this is that multiple calls to a function result in multiple edges leading from the call sites to the respective function. The result is an IPCFG with a unique source node and a unique sink node. These nodes represent the entry point of the program and its exit. For all nodes of the IPCFG, we can now directly reason about the order of execution by traversing its edges.

5.2.3 Interprocedural liveness analysis

In this section, the problem of determining the interprocedural object lifetimes is addressed. The problem is first introduced in all generality. The necessary fundament of theory is discussed. It will become apparent that the common approach towards the analysis is limited to intraprocedural control flow graphs in general. Since we would like to perform an analysis at a program wide scale, the standard algorithms require an overhaul. Namely, a modified

version of the depth first search algorithm is presented to support this. Afterwards, the specific requirements for the determination of the liveness of *code objects* is addressed.

Introduction to liveness analysis

To be able to reason about the execution history of basic blocks, an analysis technique referred to as *liveness analysis* is adopted. Originally, its purpose is to determine the uses of virtual registers in the domain of register allocation [Muc97]. This is adopted to reason about the liveness of additional objects like functions or basic blocks. For now, we will refer to such instruction code related objects as just objects O.

Definition 5.2.3. An object $o \in O$ is **live** on an edge $e \in E$ of an IPCFG $G = (V,E)$, if there exists a backpath from this edge to the point where the object was originally defined, without being redefined along this path.

The liveness attribute determines for a given edge if an object is yet to be used at a later time. This means that functions, basic blocks or instructions are executed or that variables are read. We will refer to the set of edges on which an object is live as the *live range* $live(o) \in E$. Clearly, objects can only be defined or used on nodes. We refer to $DEF(v)$ as the set of objects defined at node v and to $USE(v)$ as the set of objects used at node v.

Definition 5.2.4. $\mathbf{LiveIn}(\mathbf{v}) \in O$ is the set of objects which are live on any of the incoming edges of a node $v \in V$.

Definition 5.2.5. $\mathbf{LiveOut}(\mathbf{v}) \in O$ is the set of objects which are live on any of the outgoing edges of a node $v \in V$.

Commonly, an object is said to be *live-in* or *live-out* for a certain node if it is contained in either of the two sets just defined. With the help of the information about incoming and outgoing liveness on nodes, the live range of objects can be determined by performing a *data flow analysis* [Muc97]. We can observe the following:

- If an object is in $USE(v)$, then it must be live-in at node v.
- If an object is live-in at node v, then it must have been live-out at any of its preceeding nodes.
- If an object is live-out at node v but is not in $DEF(v)$, then it must have been live-in at v.

From these facts we can construct so called *data flow equations*. They are iteratively evaluated until a solution stabilizes.

$$LiveIn(v) = USE(v) \cup (LiveOut(v) \setminus DEF(v)) \tag{5.1}$$

$$LiveOut(v) = \bigcup_{w \in succ(v)} LiveIn(w) \tag{5.2}$$

The equations are most efficiently solved by *backward iterating*, since the set LiveOut depends on LiveIn of its succeeding nodes. As a result, the solution sets *LiveIn* and *LiveOut* stabilize significantly faster. The iterations starts with empty sets LiveIn and LiveOut. Examples to this well known algorithm can be found in [App97].

The solution to the data flow equations is conservative. Only if an object is live-out on any of the predecessors of a node, it is considered live-in at that node.

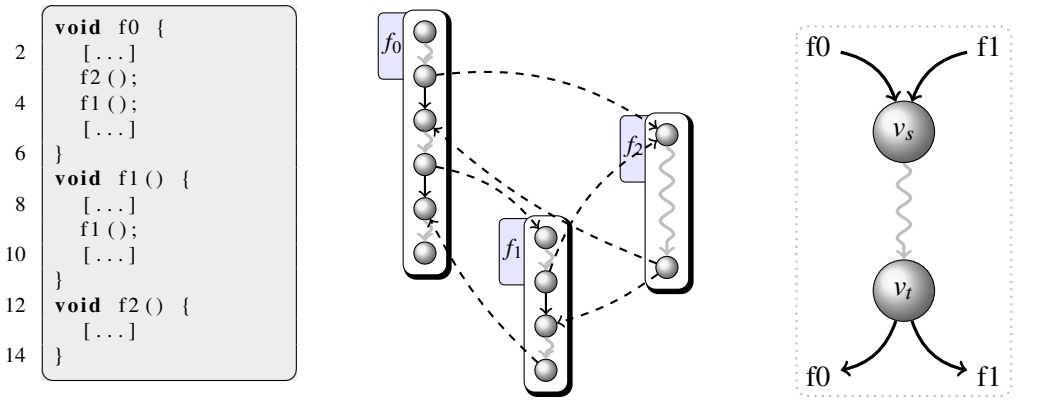
Theorem 5.2.6. *An object is considered live on an edge $e = (u, v)$ in the control flow graph, if and only if it is live-out on node u and live-in on node v.*

The precise determination of liveness is undecidable in general. That is, the liveness analysis only gives a conservative approximation about whether an object is possibly used is the future.

Interprocedural depth first search for liveness analysis

As discussed above, the liveness information on objects is obtained by continuously iterating over the control flow graph until a solution stabilizes. The classic algorithm for iterating over the nodes of a directed graph is the well-known *depth first search* (DFS) [CLRS01].

The DFS is used to traverse $\hat{G}$. Because of the special construction of call sites (definition 5.2.2), the algorithm will always enter the subgraph $\hat{G}_{f_i} \subset \hat{G}$ of the called function f_i. Conversely, if the returning node of f_i is reached, the algorithm continues with the returnee of a call site. The result is a continous traversal through all sinks and sources of all subgraphs from which $\hat{G}$ has been constructed.



(a) Calls to f_2 from two different places

(b) Call relationships of CFG

(c) Subgraph of f_2 in IPCFG

Figure 5.2: Ambiguous traversal of an IPCFG

However, this observation poses a fundamental problem that cannot be solved with a standard DFS. Figure 5.2a outlines a fragment of a source code in which a function f_2 in invoked from two different places. In 5.2b, the corresponding CFGs are illustrated. We can see that

in the corresponding IPCFG, we would be able to traverse the graph from source to sink by passing through the functions. In particular, f_2 would be traversed twice. This leads to an ambiguity as figure 5.2c shows. This subgraph is entered at its entry node but it is not possible to tell which edge to take upon exit without providing extra information on the actual traversal history. This in turn has an impact on the data flow analysis. Changes in the solution of the data flow equations are not propagated optimally. More importantly, the DFS only visits a node once. If a single function is invoked from different places, its nodes will necessarily only be traversed a single time. The traversal is stopped prematurely and the global liveness information is not propagated entirely at all.

In fact, the DFS would have to treat one and the same function subgraph as an independent instance when called from different call sites. Because a duplication of subgraphs would result in an exponential growth of the IPCFG, the DFS must be modified to take into account the different call sites when traversing. This is referred to as the *traversal context*.

A particular problem is the encoding of contexts within the DFS. The context is not only defined by the immediate call site but by the history of all call sites that have been entered before. Every call site whose one node has been visited but the other hasn't yet during DFS traversal is referred to as an *unfinished call site*.

Definition 5.2.7. Let $\hat{G} = (\hat{V}, \hat{E})$ be an IPCFG and let $\Omega = \{(u,v) | \{u,v\} \subset \hat{V} \times \hat{V}\}$ be the set of call sites. A **call string** is a sequence of unfinished call sites $\langle \omega_n, \ldots, \omega_m \rangle$ with $\omega_i \in \Omega$ along an arbitrary path starting form the source of $\hat{G}$.

The call string uniquely identifies a context for which a node is traversed. Every node $v_i \in \hat{G}$ will be represented in the *context-sensitive DFS* by a pair $\langle v_i, \langle \omega_n, \ldots, \omega_m \rangle \rangle$, referred to as a *context-dependent node*. A node is marked as finished if and only if all of its succeeding context-dependent nodes have been marked finished and the current context matches.

Listing 4 outlines the code for such a *context-sensitive DFS*. As opposed to the common DFS, call sites need to be distinguished. Every call site that is entered through the calling node causes a new context to be generated. Each traversal of a returnee node must reestablish a previous context.

The input to the context-sensitive DFS is an IPCFG and its source. In lines 2 to 6, the required data structures are set up. The set of call sites within $\hat{G}$ must be known. Moreover, the current context is denoted by a call string. This is a sequence of call sites. As in the original DFS, a set stores the nodes that have been completely processed. In this case, a node can only be finished if all of its successors are finished and the current context denoted by a call string matches its own context. Without a matching context, a node is treated as unvisited. A stack stores the nodes with their respective context that are yet to be processed. Lastly, a stack stores the previously used contexts. In lines 8 and 9, an initially empty call string is prepared which will denote the current context, and the starting node is placed onto the stack for unfinished nodes t. The algorithm will only terminate if the stack of unfinished nodes is empty (line 10). In line 11, the top of the node stack is taken. The solutions to the liveness equations (5.2.4) and (5.2.5) are updated in line 12. Lines 13 to 16 check if a call site *call* is reached. If so, a caller causes a backup of the current context and the creation of a new one. That is, every node that is looked up in the future will now depend on this

Algorithm 4: Outline of context-sensitive DFS

```
1:  {Input: Ĝ = (V̂, Ê) and start node v_s ∈ V̂}
2:  Ω: set of call sites
3:  c: call string P(Ω)
4:  f: set of finished context-sensitive nodes ⟨v, c⟩
5:  s: stack of context-sensitive nodes ⟨v, c⟩
6:  t: stack of contexts
7:
8:  push(t, ⟨⟩)
9:  push(s, ⟨v_s, top(t)⟩)
10: while s ≠ ∅ do
11:    ⟨v, c⟩ := pop(s)
12:    update_liveness(⟨v, c⟩)
13:    if v is call site ω_i caller then
14:       push(t, c)
15:       c := c ⊕ ω_i
16:    end if
17:    if v is returner then
18:       f ∪= ⟨v, c⟩
19:       c := pop(t)
20:       v := call site (ω_i = suffix(c)) returnee
21:       push(s, succs(v) × c))
22:    else
23:       if ⟨v, c⟩ ∉ f then
24:          f ∪= ⟨v, c⟩
25:          push(s, succs(v) × c))
26:       end if
27:    end if
28: end while
```

new context. All nodes that have already been visited in a different context are considered unvisited. Conversely, traversing a returnee (lines 17 to 21) causes the previous context to be restored. Here, the returner node is marked as finished, the previous context is restored and the correct out-edge is sought. There exists only one edge whose target node is member of the call site that is denoted by the suffix of the call string. The lines 22 to 26 are similar to the standard DFS. If the node hasn't been visited under the current context, mark it as visited and push its successors with the current context onto the stack.

An implication of the call site approach is that recursions cannot be encoded easily because they potentially result in infinitly long call strings. [KK08] discuss the problem thoroughly and propose a technique to deal with this problem. However, for the goal of this thesis, more complex solutions to the problem bear little benefit as the benchmarks applied are moderately sized and almost non of them is recursive.

An important aspect shall be touched at this point. It was mentioned above that it is favorable

to visit the nodes starting from the sink instead of the source because of the dependencies expressed in equations (5.1) and (5.2). Instead of modifying the algorithm, all edges within the graph can be reversed before traversal.

Definition 5.2.8. A reversed IPCFG $\check{G} = (\check{V}, \check{E})$ is constructed from an IPCFG $\hat{G} = (\hat{V}, \hat{E})$ by setting $\check{V} = \hat{V}$ and by reversing its edges, so that $\check{E} = \{(v,u)|(u,v) \in \hat{E}\}$.

The changes to the solution sets however, need to be applied as if the graph were in its original state. That is, LiveIn and LiveOut are filled according to $\hat{G}$. This is easily done as the set of nodes remains equivalent. The context-sensitive DFS requires no substantial modifications in this regard.

Summing this up, each time a yet unvisited context-dependent node is visited, the data flow equations are evaluated anew. In practice, the solutions quickly stabilize due to the backward traversal.

Interprocedural liveness for code objects

The solution we have discussed so far helps in determining the live ranges of program objects in general. For the purpose of dynamic code allocation, objects like variables need no consideration. Also, whole functions are not explicitly considered, since the data flow equations are solved on an IPCFG. The basic unit that is considered in the dynamic allocation is therefore just the basic block.

Similar to the static allocation problem, the dynamic allocation is performed by placing basic blocks into the scratchpad memory. In this case, the loading is performed during execution instead of statically choosing a set of objects that is loaded before execution. The live range gives an indication of when the dynamic loading should take place. An object that is guaranteed to be never executed again needs no further consideration. Should it have been placed into the scratchpad memory, the space it reserved can be reused. On the other hand, a decision has to be made if and at what point an object should be loaded along the execution path.

To make the data flow equations applicable to code objects in general, we make the following observation. To a code object, its point of use is its point of execution explicitly. As a consequence, its point of execution denotes the end of its live range for an acyclic path. In addition, code objects have no point of definition within the CFG. In fact, they are defined prior to execution. Therefore, the explicit live range of a code object always starts at the source node. Figure 5.3 illustrates this.

As a consequence, the IPCFG should be extended to make use of the fact that object definitions have no explicit correspondence in the graph. Also, it should be possible to load a static set of objects prior to execution into the scratchpad memory. This way an initial set of objects is already present and for small programs the duty of loading objects at execution time is entirely avoided. The actual benefit of this will become clear when the construction of the ILP is discussed in section 5.4. Here, we shall only set up the theoretic foundation. We refer to components of a graph with no corresponding objects in the program as being *virtual*.

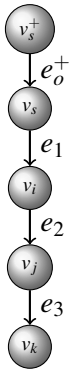
Definition 5.2.9. An **extended IPCFG** is a directed graph $\hat{G}^+ = (\hat{V}^+, \hat{E}^+)$ that is created from an IPCFG $\hat{G} = (\hat{V}, \hat{E})$ by extending the set of nodes with a *virtual node* v_s^+ so that:

$$\hat{V}^+ = \hat{V} \cup \{v^+\}$$

Also, the set of edges is extended by an *virtual edge* $e^+ = (v^+, v_s)$ so that:

$$\hat{E}^+ = \hat{E} \cup \{e^+\}$$

Figure 5.3 shows such an extended graph with its accompanying live ranges. Without loss of generality, this is valid for a reversed IPCFG as well.



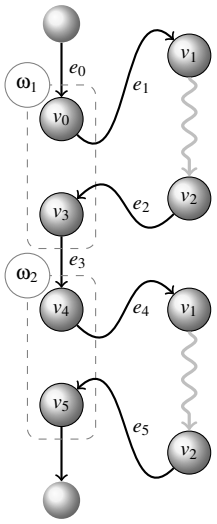
	DEF	*USE*
v_s+	$\{v_s, v_i, v_k, v_k\}$	$\emptyset$
v_s	$\emptyset$	$\{v_s\}$
v_i	$\emptyset$	$\{v_i\}$
v_j	$\emptyset$	$\{v_j\}$
v_k	$\emptyset$	$\{v_k\}$

	Live range
v_s	$\{e_0^+\}$
v_i	$\{e_0^+, e_1\}$
v_j	$\{e_0^+, e_1, e_2\}$
v_k	$\{e_0^+, e_1, e_2, e_3\}$

Figure 5.3: Extended IPCFG and live ranges of basic blocks

At this point, context-dependent liveness information on code objects can be determined fully. For each traversal of a subgraph $\hat{G}_{f_i} \subset \hat{G}$ of a function f_i, independent live ranges are available after performing the data flow analysis. However, this context dependence of the results poses yet another problem. For every allocation decision that results in the dynamic loading of basic blocks during execution, explicit copy routines need to either be invoked as functions or injected into the original code. Every call site implies an independent instance of a function from the perspective of the data flow equations. In fact, there only exists a single instance represented as the subgraph $\hat{G}_{f_i}$ within $\hat{G}$. The consequence is that allocation decisions for one and the same subgraph are potentially different because the solution of the data flow analysis suggests so. Figure 5.4 demonstrates the result of a possible liveness analysis. In an IPCFG that encompasses the nodes v_0 to v_5, a subgraph that represents a function is encoded which is exemplary starting with v_1 and ending with v_2. This function is invoked from two call sites, namely ω_1 and ω_2. Due to the distinction of contexts, the live range of the nodes representing basic blocks is determined as shown in the table. Since a call site induces another context, the live ranges for the same nodes v_1 and v_2 are different because they are determined with different contexts and are therefore potentially distinct nodes from the perspective of the data flow analysis.

The context-sensitive data flow analysis was motivated by the fact that an explicit instantiation of multiple subgraphs per function should be avoided. However, after having performed the analysis, the results must be transformed, so that they become context-free. That is, the results must be applicable to an IPCFG - which has no notion of traversal contexts.



	Live range
$v_0\langle\rangle$	$\{e_0\}$
$v_1\langle\omega_1\rangle$	$\{e_0, e_1\}$
$v_2\langle\omega_1\rangle$	$\{e_0, e_1, \dots\}$
$v_3\langle\rangle$	$\{e_0, e_1, \dots, e_2\}$
$v_4\langle\rangle$	$\{e_0, e_1, \dots, e_2, e_3\}$
$v_1\langle\omega_2\rangle$	$\{e_0, e_1, \dots, e_2, e_3, e_4\}$
$v_1\langle\omega_2\rangle$	$\{e_0, e_1, \dots, e_2, e_3, e_4, \dots\}$
$v_5\langle\rangle$	$\{e_0, e_1, \dots, e_2, e_3, e_4, \dots, e_5\}$

Figure 5.4: Example: Context-dependent live ranges

The liveness in general is calculated by making use of a conservative strategy (theorem 5.2.6). If a path exists between an object's definition and its use, it is considered live. A similar approximation is applied to the context-dependent liveness information to make it context-free.

Theorem 5.2.10. *For a given IPCFG $\hat{G} = (\hat{V}, \hat{E})$, context-depent live ranges are transformed into context-free live ranges by conjoining the sets LiveIn and LiveOut as defined in 5.2.4 and 5.2.5 for all contexts of a specific node in $\hat{G}$:*

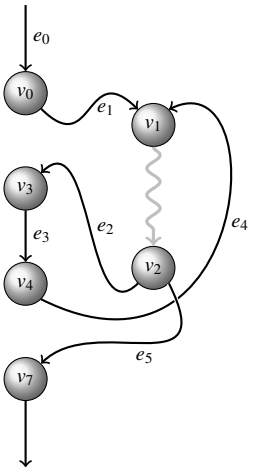
$$LiveIn_{ctxfree}(v) \cup\!= LiveIn(\langle v, \omega\rangle) \qquad \forall\omega \in \Omega, \forall v \in \hat{V} \tag{5.3}$$

$$LiveOut_{ctxfree}(v) \cup\!= LiveOut(\langle v, \omega\rangle) \qquad \forall\omega \in \Omega, \forall v \in \hat{V} \tag{5.4}$$

In the following, we will refer to the context-free sets simply as *LiveIn* and *LiveOut*. The liveness per edge is still determined independently of this according to theorem 5.2.6.

The conjunction of context-dependent information has the consequence that if a node external to a function has been considered live at one point in the execution history, it will be considered live in every execution of this function although for some cases this assumption might be overly conservative. The conjunction expresses that a possibility of execution for a code object exists.

As we will see in section 5.4, the transformation is safe because the ILP we will construct would not allow for redundant allocation decisions unless an advantage in terms of WCET reduction would arise from it. In fact, this never happens because only basic blocks external to a given function can be falsely considered live within its body but there is no advantage by loading an external block within the function as opposed to loading it before or after the function has been executed. The conservative estimation of liveness does not lead to inefficient decisions.



	Live range
v_0	$\{e_0\}$
v_1	$\{e_0, e_1, e_3\}$
v_2	$\{e_0, e_1, \ldots, e_3, e_4\}$
v_3	$\{e_0, e_1, \ldots, e_2, e_4\}$
v_4	$\{e_0, e_1, \ldots, e_2, e_3\}$
v_5	$\{e_0, e_1, \ldots, e_2, e_4, e_5\}$

Figure 5.5: Example: Context-free live ranges

The example in figure 5.5 illustrates the result from not distinguishing contexts and thus from transforming the live ranges that have been originally considered in figure 5.4. The graph now reflects the actual layout which is used to construct the ILP model in section 5.4. As can be noted, the live ranges for the node v_1 and v_2 now reflect the combined information of both contexts that have originally been distinguished. Therefore, the nodes are also considered live on edges e_3 and e_4.

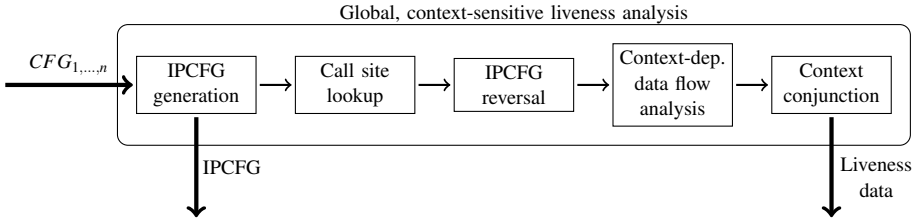


Figure 5.6: Workflow of global liveness analysis

In this section, we have discussed the problems that arise for solving the liveness problem at an interprocedural scale. After having constructed an IPCFG and after determining the liveness as just discussed, the prerequisites for solving the dynamic allocation problem have been fulfilled. Globally, an approximation of the live ranges of basic blocks can now be determined. It was shown how existing algorithms could be extended to conveniently solve this problem. Figure 5.6 illustrates the workflow.

5.3 Towards a dynamic allocation

Now that a foundation of information on the execution behavior of basic blocks has been determined, a motivation for the solution to the dynamic allocation problem shall be given.

To solve the static allocation problem a single ILP model has been generated. In section 5.1, it has discussed that ILP is also a popular way to encode the dynamic allocation problem.

Many solutions deal with allocations that are aimed at energy reduction. As was already presented in chapter 4, for code objects, the selection of sets of consecutively executed objects is an important factor to achieve viable results. [Ste02] introduces *multi basic blocks* which are essentially elements of a power set of the basic blocks of a function. Equally to *traces* presented in [Ver06] they aim for minmizing the impact of moving objects between the memories. Both techniques are aimed at energy reduction. The problem of the existing solutions in the context of this thesis is that the WCEP is only implicitly modeled. Depending on the decisions in the ILP model, it can dynamically change (refer to figure 4.6).

The ILP model in [Ver06] is derived from the original work of [GW96], which propose an ILP model for solving the register allocation problem. Indeed, the problem of dynamic allocation of objects to different memories is closely related the problem of register allocation. In the particular case of this thesis, a problem similar to the global register allocation problem for non-uniform registers [KW98] is to be solved. The global register allocation problem is NP-complete [HP02]. For all energy- and WCET-related works, a (locked) cache or a scratchpad memory serve as a pool of non-uniform registers.

In [AG01], [DP07] and [Ver06] the problem is identified to consist of two subproblems. The selection of objects for the allocation and the actual placement in the target register or memory. In the latter work, a unified ILP model is presented that solves both the problems but it is pointed out that for large problems the solving time can be significant[1]. Therefore, a two-step approach is proposed. The ILP is concerned with the selection of objects, and an algorithmic solution to the address assignment problem is obtained in a second step. It is pointed out that a simple *first fit* [JW99] strategy works well in practice. Due to these experiences, a similar approach will be taken. In section 5.4, an ILP is presented that solves the dynamic allocation problem as far as the memory selection is concerned. In section 5.5, a postprocessing step is presented that is concerned with the actual placement within the memories.

It should be noted that although [DP07] present a solution to the WCET-directed dynamic allocation problem for data, the solution in this thesis is aimed at the allocation of code objects. As we will see, this poses a new set of problems that have not been addressed thoroughly in other works. Moreover, the named work relies on a repetitive evaluation of the WCEP explicitly. In chapter 4, it was motivated why this approach is insufficient. The concept of implicit WCEP consideration is applied to the dynamic allocation problem here, which is an attempt to provide an enhanced strategy as opposed to existing works.

5.4 ILP model

Referring to what has been discussed in the previous section, the problem of dynamically allocating code objects is similar to the global register allocation problem. There, a set of

[1] Without referring to a specific benchmark, a duration longer than three weeks was observed on a 1.3GHz SPARC machine.

symbolic registers is to be mapped into a smaller set of physical registers. If no free physical register is available at a certain point of execution, the contents of the symbolic registers are kept in the main memory. In a load/store architecture, this means that accessing data that has been *spilled* into main memory needs to be spilled back into a register before usage. Obviously, it is important to the overall performance of a program at which point in time this spilling takes place, should the mapping fail. Spilling into a register close to its use runs the risk to place the spill code into a loop body. Repetitive loading could result. Spilling distant to the point of use reserves a register over a longer period of execution time. It could have been used for other purposes meanwhile.

A similar problem arises for the dynamic scratchpad allocation. The objects should be spilled into the scratchpad memory prior to execution. Loading too early reserves the scarce space. Loading too late could result in redundant spill actions.

The allocation of code objects requires more attention than for data objects. Code objects are interdependent. Arbitrarily distributing the code objects without taking into account their execution order is a problem which further restricts the solution space for the dynamic allocation. This is because jumps between basic blocks can severely influence the execution performance. The same issues as discussed in chapter 4 arise from this.

The ILP model presented in this chapter is an extension to the one presented in chapter 4. During the following discussion, it will be referred to as a *structural model*. The required modifications to this model are defined in section 5.4.2. The same problems concerning the weighting of basic block execution costs and the grouping of basic blocks are to be solved in the dynamic case as well. The extension we will present in section 5.4.1 provides a *spill model* which allows to solve the problem of spill point determination with regard to the global program structure. Both models are combined into a single ILP.

5.4.1 Definition of the spill model

The spill model encodes the problem of finding an appropriate position in the program at which a code object should be loaded into the scratchpad memory. It will be presented later how to judge if an object should be loaded at all. An important observation at this point is that code objects are never spilled from the scratchpad memory back into the main memory because instructions are considered to be a static resource that cannot change their "contents" as opposed to general data.

To load an object, the original program requires modification. Instructions need to be emitted that either transfer the object directly from one memory into another or a function is to be called that performs this task. All spill code is always located within the main memory. This is referred to as *spill code*.

From the interprocedural data flow analysis presented above, we can obtain *predicates* which denote the liveness state of a basic block.

$$P_{flow}(e,v) = \begin{cases} CONT & \text{if basic block of v is live on edge } e=(u,w) \\ USE & \text{if basic block of v is live on edge } e=(u,v) \\ NOFLOW & \text{if basic block of v is not live on edge } e \end{cases} \tag{5.5}$$

The decision for when a spilling should take place is obviously bound to edges and nodes whose predicate P_{flow} is $CONT$ or USE. The decisions themselves will be bound to edges rather than nodes in the IPCFG.

For every edge $e_j \in \hat{E}$ in the IPCFG $\hat{G} = (\hat{V}, \hat{E})$, we can make the observation that a node $v_i \in \hat{V}$ has already been loaded into the scratchpad memory upon traversal or we could decide to load it at the current edge. Let mem_{main} represent the main memory and mem_{spm} represent the scratchpad memory. We can define the *location attribute* x_j^i as:

$$x_j^i = \begin{cases} 1 & \text{if the block of node } v_i \text{ is present in } mem_{spm} \text{ on edge } e_j \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

In addition, the *load attribute* y_j^i denotes whether a loading action should be performed and is defined as:

$$y_j^i = \begin{cases} 1 & \text{if the block of node } v_i \text{ is to be loaded on edge } e_j \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

Both x_j^i and y_j^i are used as binary decision variables within the ILP model. If the decision to perform a load has been made on an edge e_j, the consequence is that the information of the allocation to mem_{spm} has to be propagated througout the succeeding edges.

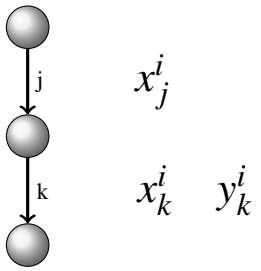


Figure 5.7: Propagation of location attributes

For the edges e_j and e_k as shown in figure 5.7, we can define the location and the load attributes to be related as:

$$x_k^i = x_j^i + y_k^i \qquad \forall v_i \in \hat{V} \tag{5.8}$$

If a node is located in the scratchpad memory on an arbitrary edge, then it must have been located in that memory on the preceeding edge or it must have been loaded on the current edge.

These equations reflect the ILP constraints for simple edge dependencies within an IPCFG. To make them viable to arbitrary graphs, certain refinements have to be made. In addition, the loading decision can only be made on living objects. If a basic block is guaranteed to be never executed again, it would be useless to load it into the scratchpad memory.

[GW96] made a general observation for the placement of loading decisions. Although it is aimed at the problem of register allocation, the statement is still valid for the specific

problem presented in this chapter. We refer to a *merge node* if its number of in-edges is greater than one.

Theorem 5.4.1. *For an optimal spill code placement it is sufficient to allow the load attribute y^i_j to be equal to one for each node v_i on edge e_j that satisfies the following constraints:*

- *The flow predicate $P_{flow}(v_i, e_j)$ is USE or CONT.*
- *The edge e_j leads to a merge node.*

The proofs to this theorem can be found in [Ver06] and [GW96]. The implicit consequence from this is that loading should not take place at nodes with a number of out-edges greater one.

A self-loop is a node with an outgoing edge whose target is the node itself (figure 4.2c). Such edges are to be ignored. For all other loop constructs, we may even allow loading on back edges. For large intervals that are bridged by such an edge, loading should not explicitly be denied.

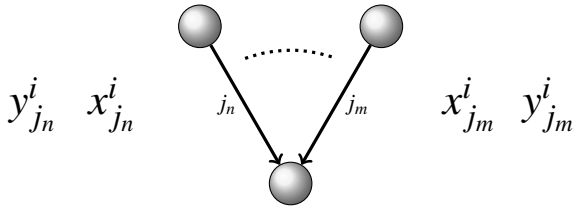


Figure 5.8: Attribute propagation on merge nodes

Another issue are branches and joins within the graph. In theorem 5.4.1, the possible spill code placements have been restricted in general. The following equation enforces the requirement for edges to merge nodes. In addition, it assures consistency of the location attributes:

$$y^i_j \leq x^i_j \qquad \forall e_j \in \{e_{j_n}, \ldots, e_{j_m}\}, \forall v_i \in \hat{V} \tag{5.9}$$

$$x^i_{j_n} = \cdots = x^i_{j_m} \tag{5.10}$$

This is illustrated in figure 5.8. Equation (5.9) ensures that for all in-edges to a merge node a decision to load also forces the location attribute to be set. In other words, if a loading takes place on a certain in-edge, then mark the object as loaded on that edge as well. Equation (5.10) enforces that all location attributes for the same object on all in-edges is equivalent. This means, if an object is located in the scratchpad memory at one of the incoming edges of a merge node, then it must be assigned to the scratchpad memory or loaded on each of the remaining edges. This ensures consistency, as an object will always be in the same location no matter which path has been taken to reach the merge node.

Up to this point, the spill model allows for a loading decision at an arbitrary point along an object's live range and the propagation of this information throughout the graph. An

important restriction is yet to be defined. An object can only be spilled into the scratchpad memory if there is enough space left. It was mentioned earlier that we will not define the position of the object within the memory. Instead, only the decision for the location of the spilling is considered. For each loading decision, the accumulated size of objects within the scratchpad memory may not exceed its overall size. Again, the dynamic growth of objects requires consideration. Specifically, should a basic block be placed in a different memory than its predecessors or its successors, its instructions may require modification so as to restore the correct executability. This problem has been thoroughly discussed in section 4.4.2.

Let the dynamic size s_{dyn} be the growth of basic blocks as defined in equation (4.27) and let $s(v)$ be their original size. Also let $\hat{V}_{live}$ be the set of nodes whose flow predicate is either USE or $CONT$. Then the ability to load a basic block into the scratchpad memory is restricted by the overall size of the target memory mem_{spm} denoted by $size(mem_{spm})$ so that:

$$\sum_{v_i \in \hat{V}_{live}} x_j^i \times (s(v_i) + s_{dyn}(v_i)) \leq size(mem_{spm}) \qquad \forall e_j \in \hat{E} \tag{5.11}$$

For all live objects, the size restriction applies. Implicitly, this means that objects that were loaded into mem_{spm} but whose live range ends at a certain point, will not be considered anymore in this constraint. Its reserved space is therefore implicitly freed and can be reused. As opposed to writeable objects like those representing variables, they need not be stored back into the main memory. An important difference to the static allocation at this point is that basic blocks in the same memory also require code modifications because they can be arbitrarily distributed within the scratchpad memory. It is assumed that every blocks ends with an explicit jump instruction. This becomes clear when the memory assignment problem is addressed below.

The spill model is now complete. Loading of objects and the restriction of memory sizes have been modeled. Still, it remains to decide if a loading should take place at all. In the next section, the combination of this spill model with a structural model similar to that defined in the previous chapter will be presented.

5.4.2 Definition of the structural model

In chapter 4, a binary decision variable a_i denoted if an object is to be statically placed into the scratchpad memory. Similarly, the location attribute x_j^i denotes this for every edge e_j. The connection between the two models is the dynamic location attribute x_j^i which replaces the static placement variable a_i.

Trivially, it can be observed that a basic block only contributes to the WCET if it is executed. The contribution depends on where it is located at the time of its execution. The costs in the structural model for a given node v_i depends on the location attribute x_j^i with $e_j = (u, v_i)$. In other words, it depends on the location of the object at the point its flow is classified USE according to equation (5.5).

The structural model shall not be discussed in this chapter in all detail again. Only the fundamental differences to the original model from section 4 will be discussed. All conclusions

that have been drawn in the previous chapter concerning the structure and the modeling of control flow graphs equally apply here.

In equation (4.7), the fundamental flow constraint has been defined. It models the contribution of a given basic block to the overall WCET by considering the costs of the path from the node to the sink of the control flow graph and its own costs. A basic block's own costs representes its isolated WCET, depending on the fact into which memory it has been placed statically.

Spill costs

From the spill model defined in section 5.4.1, we can obtain the dynamic location of a block. As we have discussed, only the location right before its execution is relevant for these costs. As was discussed earlier, the decision to move an object from one memory into another requires the introduction of spill code. This copying also contributes to the overall WCET. However, the costs are added to the block which follows the edge the decision for spilling was made on. Put differently, if the transfer of objects $v_n, \ldots, v_m$ is decided on an edge $e_j = (u, v)$, then the costs caused by the introduced spill code are added to the node v. This is reflected by the function $c_{spill}(v_j)$ which is defined as:

$$c_{spill}(v_i) = \sum_{v_k} c_{copy}(v_k) \times y_j^k \qquad e_j \in \text{ in-edge}(v_i) \tag{5.12}$$

For a given node v_i, the total spill costs is the sum of the costs that is caused by copying objects. An object v_k is loaded on edge e_j if and only if its corresponding load attribute on this edge y_j^k is set. These costs do not change for nodes v_i of in-degree greater two. The costs for the very same node is only applied once.

The copy costs per node $c_{copy}(v_i)$ are determined by estimating the amount of program code to be transferred and also depend on at which point during execution it is performed. This requires a more detailed discussion. In section 5.6, the spilling and the cost estimation are investigated in detail. Implementation specifics shall not be discussed at this point.

Jump costs

Another important change is applied to the equations that model the jumps between basic blocks. In section 4.4.2, a model was presented in which jump instructions are classified according to whether they are conditional, unconditional or if an explicit instruction existed at all. The three classifications are shown in figure 4.14.

The motivation behind this was to penalize changes in the instruction code. Given that two basic blocks are located in two different memories, the original jump instructions are insufficient to encode the jump over large address space intervals. Moreover, basic blocks that were connected by an implicit jump but were selected to reside in different memories have an even greater impact on the execution performance.

By binding penalty values to the allocation decisions in the static allocation model, arbitrary distributions of basic block can be prevented. The jump penalties were presented in equation (4.14).

It was defined that the partial order of basic blocks is an invariant property to the static allocation. This is important because the assumptions regarding code layout (branch prediction in particular) should not be changed. The arbitary placement of basic blocks within a memory can severly impact the execution performance (see section 3.3).

In the case of the dynamic allocation problem, the order of basic blocks cannot be strictly guaranteed. This is because the placement of the objects within the scratchpad memory changes over time and ideally, all unallocated space should be reused in general. Therefore, the order of basic blocks cannot be fully guaranteed. But similarly to the jump penalization for the static allocation, an approximate model can be constructed which should result in a profitable optimization. In fact, the original penalization can be extended. The limitation which draws it unsuitable to the dynamic allocation problem is that it is only distinguished into which memories two or more objects have been placed. This is sufficient since the order is guaranteed.

Without being able to know the resulting order upfront, the placement within a single memory requires consideration, too. The order and the actual placements are not known. However, we can indirectly achieve an allocation behavior in which not keeping the order is penalized. Given two basic blocks are connected by an implicit jump within the same memory. Placing them into arbitrary places requires a modification of the instructions according to section 4.5.2. This necessarily causes an overhead as discussed. If both blocks were connected by an explicit jump prior to replacement, we can expect that no additional overhead is caused because only the target operands would require a modification in general. On the TriCore1 architecture, a jump between memories always requires an indirect jump which demands for multiple additional instructions. In comparison, an explicit jump at most requires an exchange of a single instruction. For the purpose of this model, the overhead between single 16bit wide and 32bit wide jump instructions is neglected.

The result from the ILP model only determines whether objects are placed into the scratchpad memory. The exact location is determined in a postprocessing step. In this step, every object ever loaded is assigned a fixed address to which it is always spilled. The original jump penalties cause a selection of objects favoring a consecutive control flow. But this is insufficient in the dynamic case. Although the location attribute according to (5.6) suggests that consecutive blocks have been placed into the same memory, it does not mean they are loaded together. As a consequence of being loaded at different points in time, the placement in the postprocessing step might separate the objects.

Figure 5.9 illustrates the problem. The example shows a fragment of a control flow graph on the left. On the right, a possible memory allocation for the scratchpad memory is illustrated. At time T_0, only three objects fit into the scratchpad memory. The basic block of node v_d cannot yet be placed. The allocation algorithm we will discuss in section 5.5 in detail takes care of keeping the original order execution of objects if possible. However, at time T_1 the missing block can be loaded into the scratchpad memory. But the placement of its successor does not allow to restore the original order. The block of node v_d is placed on top of the

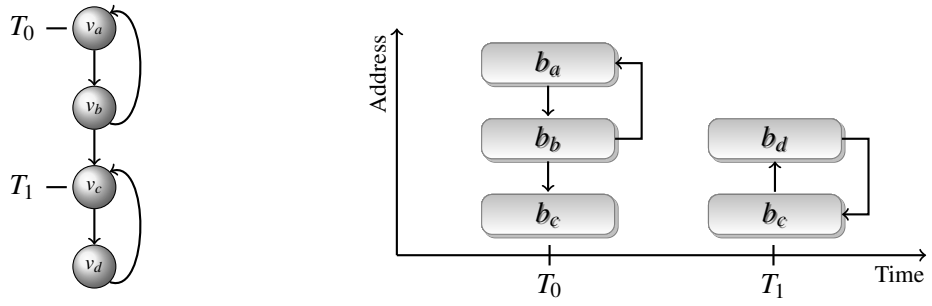


Figure 5.9: Improper object placement

other in this example. In fact, it could be placed anywhere in the memory. The consequence is that the displacements for the instruction pointer due to jumps are now reversed in their direction. In section 3.3, the branch prediction of the TriCore1 processor was presented. As can be seen, the result of the allocation in this example is a worst-case scenario, because the conditional jump of the loop is statically considered not taken.

To prevent this from happening, a loading of consecutive basic blocks at the same points in time must be favored. Put differently, the loading at different points should be penalized. The worst-case expectation from loads at different times is that implicit jumps need to be converted into explicit jumps. Above, we have already discussed why only distinguishing between implicit and explicit jumps is sufficient as an approximation.

The load penalty is applied in conjunction to the jump penalties. Both cause the favoring of larger groups of objects. The objective of the jump penalties is to favor structural grouping. The objective of the load penalties is to favor temporal grouping.

$$J^i_{Load} = \begin{cases} (y^i_k \otimes y^j_k) \times P_U & \text{for implicit jumps} \\ 0 & \text{otherwise} \end{cases} \quad \forall v_i \in \hat{V}, \forall e_k \in \hat{E} \tag{5.13}$$

The variable J^i_{Load} represents the penalty applied, when consectuive objects are loaded at different points in time. The node v_j is the implicit successor of v_i. This involves conditional jumps as well. The loading of objects is denoted by the variable y^i_k. It is set if the object of v_i is loaded on edge e_k. If both are not loaded simultaneously, we have to apply the penalty of an *uncritical* jump P_U as we must assume that an explicit jump will be required which is limited to the current memory. This is modeled by applying the *XOR* operation ($\otimes$) with the load attributes as its operands.

The jump penalty function c_{jump} is now defined as

$$c_{jump} = J^i_{Load} + \begin{cases} J^i_{Implicit} & \text{if the jump case is } \textit{implicit} \\ J^i_{Unconditional} & \text{if the jump case is } \textit{unconditional} \\ J^i_{Conditional} & \text{if the jump case is } \textit{conditional} \end{cases} \tag{5.14}$$

The other constants are equivalent to the ones presented in section 4.4.2.

Base constraint

The basic flow constraint for the dynamic allocation problem is therefore defined as:

$$\begin{aligned} w_i \geq w_{i+1} + c_{main}(v_i) - gain(v_i) \times x_j^i \\ + c_{jump}(v_i) + c_{call}(v_i) + c_{spill}(v_i) \end{aligned} \tag{5.15}$$

The fundamental idea of how a total WCET is estimated remains equivalent to what has been discussed for equation (4.25). The gain function is the same as defined in equation (4.6). Instead of depending on a static decision, a gain can only occur when the corresponding basic block is actually located in the scratchpad memory upon execution. This is denoted by x_j^i as defined in (5.6). In the previous chapter, the problem of jump penalties has been throughly discussed . Again, we can make the same assumptions about the penalization and reuse the estimation function j_{jump} (equation 4.9).

All the presented equations can be directly incorportated into an ILP model. The overall optimization objective is identical to the one defined in chapter 4. That is, the objective is to lower the estimated global WCET. The structural and the spill model in combinaton describe the dynamic allocation problem as far as the assignment to different memories is concerned. The ILP can now be solved. However, the ILP result requires further processing.

5.5 Postprocessing

The ILP model we have constructed up to now is capable of solving the memory assignment problem for the dynamic allocation. As was motivated in the introduction, the solution process is divided into two steps. Now that the load and location attribute variables have values assigned, it is required to determine memory addresses the basic blocks are to be placed at.

5.5.1 Memory allocation principles

Basic aspects of memory allocation shall be briefly presented in this section. For this purpose, we abstract from the actual result of the dynamic memory assignment problem and refer just to objects that need to be placed into a memory.

Finding a place in an address space for an object of a given size is generally referred to as *memory allocation*. This process assigns a specific start address to the object, which is why it is also referred to as a *memory assignment problem*.

The solution provided by the previously presented ILP model selects distinct places within the code where a loading of objects from the main memory into the scratchpad memory should take place. Therefore, for a given point in time, a well defined set of objects is expected to reside in the scratchpad memory. Figure 5.10 illustrates this exemplary. The difference to the static allocation becomes apparent when compared to figure 4.8. In the figure here, a static set of spill points $S_0, \ldots, S_3$ has been selected by the ILP. Only at these

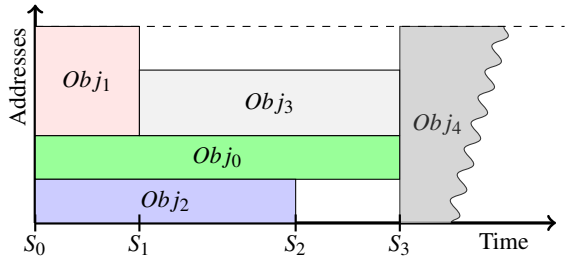


Figure 5.10: Memory layout of dynamic allocation

points, the loading of objects into the scratchpad memory is possible. Over the course of time, different objects can be spilled. When the live range of an object ends, the space it occupied can be reclaimed. Objects are never transferred back into the main memory once they have been placed into the scratchpad memory. But they can be overwritten.

The ILP model enforces an upper limit on the total memory size. The decision for loading an object depends on whether the sum of object sizes[2] is smaller or equal to the memory size (equation 5.11). The task of the postprocessing step presented in this section is to find appropriate spaces within the address space over the course of execution time.

In [JW99], the problem of allocating memory space is addressed thoroughly. If all memory objects were of the same size, then an efficient algorithm can be constructed for allocation. However, the problem of allocation for non-uniformly sized objects is NP-complete. The primary problem is *fragmentation*. Figure 5.11 shows the problem.

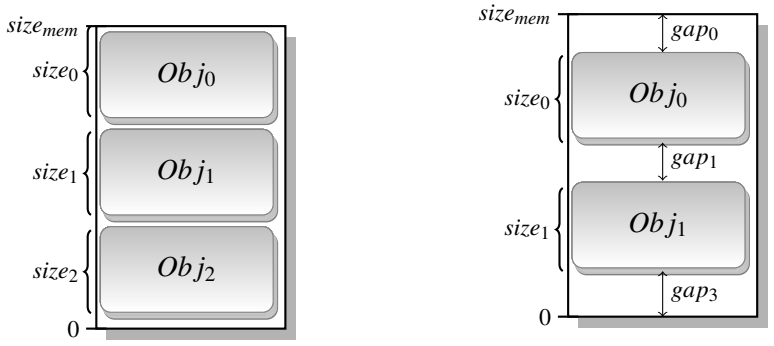


Figure 5.11: Fragmentation upon allocation

Although the sum of object sizes suggests that all objects fit into the target memory (figure 5.11a), the fact that allocations and deallocations happen frequently can lead to situations

[2]Dynamic growth is neglected in this discussion.

where no suitable space in the address space can be found (figure 5.11b).

5.5.2 Address assignment for dynamic solutions

A major problem we encounter is the fact that the assignment solution is based on basic blocks as its basic unit of allocation. It is necessary to form larger units that are handeled atomically. The selection must be performed so that the number of cross jumps between basic blocks is minimized. The basic blocks to be placed into the scratchpad memory are only known after solving the ILP. At best, larger units can be selected while the ILP is solved by encoding this problem as part of the model. But due to the implicit representation of the WCEP, this is a complex task. In the structural part of our ILP model, jump penalties are applied to basic blocks to minimize the separation of connected ones. In the solution to the static allocation problem, this proved to be an effective way to group basic blocks. We make use of this fact for the address assignment.

So, instead of preselecting groups of basic blocks, an attempt is made to post-select them. This fact makes the solving of the overall problem in two successive steps mandatory after all. It is not easily possible to encode this into the ILP. It should be noted that [Ver06] also suggested an integrated address assignment directly within a single ILP model. This resulted in remarkably long solving times in some cases, as was alluded earlier.

To perform a viable address assigment, groups of basic blocks need to be maximized.

Definition 5.5.1. A **basic block group** is a maximal set of basic blocks represented by connected nodes within a control flow graph.

A group is therefore not a strongly connected component. It is sufficient to find a maximal set of sequentially executed basic blocks.

Due to the application of jump penalties, the basic blocks tend to be allocated as groups already. These groups can be found by applying a depth first search to identify connected components.

Now that the basic block groups have been identified, it is important to keep them ordered. The order should be equivalent to that in the original program.

In figure 5.12a, a grouping is demonstrated. From a given control flow graph, the ILP determines a set of objects to be moved into the scratchpad memory. From the basic blocks represented by the marked nodes in the graph, groups are determined. The placement in the scratchpad memory then only takes into consideration groups. As is illustrated in figure 5.12b, the corresponding basic blocks will only be spilled into the scratchpad as these groups. The task of a address assignment is to determine locations for the groups to which they can be transferred to at execution time.

The placement of basic blocks out of a single memory into another one requires an instruction adjustment as was already thoroughly discussed in section 4.5.2. Such correction inevitably leads to a growth of the basic block size. During memory assignment, the growth was estimated. Now that the address assignment takes places, this estimation also requires consideration.

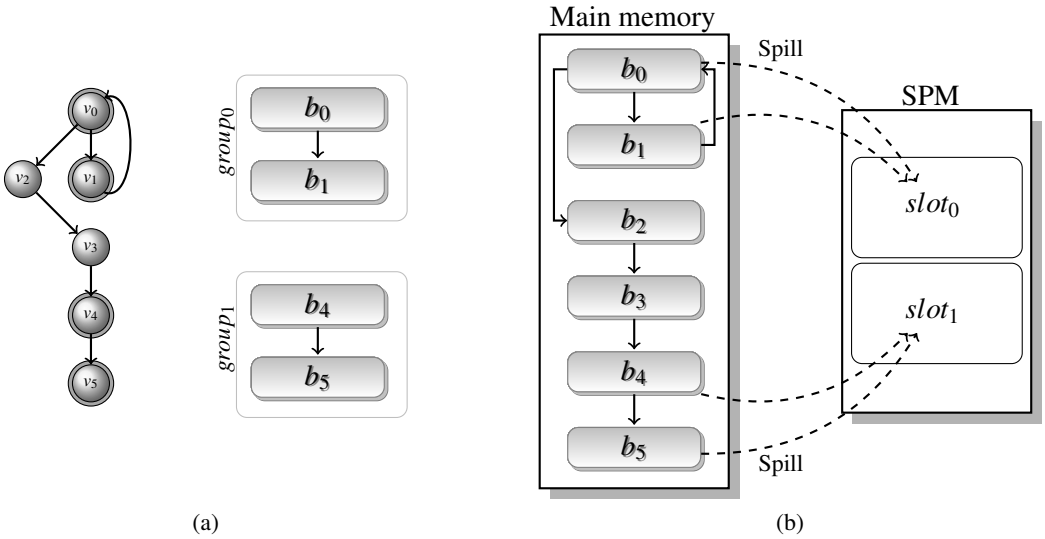


Figure 5.12: Grouping of basic blocks

The strategy to assign addresses to the basic block groups is to make use of a simple linked list allocator [Knu73a] with a first fit strategy. In [JW99], different allocation techniques have been investigated and it was concluded that such a simple approach already leads to good results. This has been affirmed by [Ver06].

Repetitive changes in the memory layout due to spill decisions at different points can lead to fragmentation of the memory. This leads to the problem of not being able to allocate a suitable space. The basic block groups are therefore allowed to drop single elements instead of reassigning the complete group back to main memory if the allocation fails.

Although the allocation is dynamic at execution time, the decisions for when an allocation should take place is statically determined at compile time. Moreover, the ILP model enforces that when a basic block is executed, it will always be present in the same memory. From this observation, an algorithm can be developed that statically assigns addresses to basic block groups. The blocks will always be executed from the same location.

Listing 5 presents the algorithm for solving the address assignment problem. The algorithm takes as input the IPCFG and the set of decision variables from the ILP. The former includes the information on the global liveness, the latter contains information on the spilling decisions. In lines 3 and 4, the relevant nodes on the current edge are extracted. They must be living and the spilling for the particular node must happen on the current edge. From this set of nodes, the groups are determined (line 5). A node is assigned to a group just once. That means that once a group has been established. it remains throughout the allocation. This is important to note because on different edges different group members are possible. The memory map is refreshed on line 6. For each edge, a different memory usage of the scratchpad memory is encountered depending on what allocations have already been performed. For all groups that have just been determined (line 7), slots in the address space of the scratchpad memory must be assigned. Groups that already have an address assigned

Algorithm 5: Outline of address assignment algorithm

```
1:  {Input: IPCFG Ĝ, ILP decisions}
2:  for all egdes e ∈ Ê do
3:    L := living nodes(e)
4:    L' := loaded nodes(e, L)
5:    BG := determine groups(L)
6:    M := update memory map
7:    for all bg ∈ BG do
8:      if bg ≠ empty ∧ bg has no slot in M then
9:        bg' := estimate dynamic growth(bg)
10:       s := find slot first fit(M, bg')
11:       if no slot found then
12:         remove element from bg
13:         retry
14:       else
15:         update memory map(M)
16:       end if
17:     end if
18:   end for
19: end for
```

are omitted (line 8). From the discussion in section 4.5.2, it is clear that basic blocks might require a modification of instructions. In line 9, for each group a safe growth estimation is performed. For this estimated size, a slot in the memory map is seeked (line 10).

If, due to fragmentation, a slot could not be found (line 11), an element is taken from the current group and is not considered for allocation into the scratchpad memory. The current heursitic selects the element with the lowest costs. After removing this element from the group, the allocation is attempted again.

Otherwise (line 15), the internal memory map is updated to reflect the changes. Once a slot has been assigned to a group, its address is fixed throughout all edges of the graph.

Since the addresses are fixed, this also allows to adjust the jump instructions of all basic blocks according to the technique discussed in section 4.5.2. Every time a basic block is executed, its context as far as succeeding basic blocks are concerned is always the same. Therefore, a single jump adjustment would suffice to restore the executability of the program if these execution contexts would be taken into account separately. In practice, the problem of restoring the executability is more complex. The problems related to this issue are discussed in section 5.8. In the following, the generation of spill code is addressed.

5.6 Spill code

Loading the basic block groups into the scratchpad memory requires a modification of the original program code. Two approaches can be taken. Either a call to a function can be emitted, or the required instructions can be provided as inlined instructions. Invoking an external function for the transfer has the advantage that the overall overhead is low and the influence in the original program code is minimal. However, the WCET analysis requires that for each function call the worst case is assumed. A large overestimation would result. Emitting instructions into the existing code causes a greater overhead in terms of program size, but a more precise analysis is possible. In this thesis, the latter approach is the default. The actual implementation also allows for generating a separate copy function, however.

The objects for the transfer, the points at which they need to be transferred and the target addresses are already known. Since the basic blocks have been grouped like shown in the previous section, the transfers are performed on program code that usually encompasses multiple basic blocks.

For each edge $e = (u, v)$ on which a loading decision has been made, spill code is emitted between the two basic blocks in the control flow. If an edge leads from a call site to a function, then the code can safely be emitted as the first instructions of the called function. If the edge represents a return from a function, the same applies.

An issue with the spill code placement is that all modifications to the program code have to consider that the register allocation has already been performed. But for the new spilling instructions, a set of registers needs to be available. Because of this, every spill code starts with a prologue sequence that saves the necessary registers onto the program stack and ends with an epilogue that restores the original registers from the stack. A special case are function call edges. Since the register allocation is limited to functions, the spill code executed first in a function can safely use some registers except the ones that were used for passing arguments. Equally, this holds true for function returns. For each edge which demands for spill code generation, only a single pair of prologue and epilogue code needs to be generated.

Figure 5.13 outlines the workflow that is performed per edge. If there are no spill decisions on the edge, nothing has to be done. Otherwise, we have to distinguish if the edge is connected to a call site as either a calling, a returning or a virtual edge. If this is not the case, explicit register saving needs to be performed. The default behavior for the spill code generating routines is to emit inlined code. However, the possibility for generating an external copy function has also been provided in the generating component of the optimization. Instead of inlined code, the respective function calls would be emitted. If the code is to be inlined, it is specifically generated for the task on this edge. If the amount to be transferred is not a multiple of the machine word[3], copying is started with smaller units until the amount is properly aligned. Then, we can transfer the remaining data with a small instruction footprint because the TriCore1 provides instructions that make this particularly efficient. For large amounts of data, a tight loop is generated. This makes use of the Tricore1-specific loop pipeline (refer to section 3.3). After one iteration, the loop can

[3] On TriCore1, this is a 32bit wide value. Refer to section 3.3.

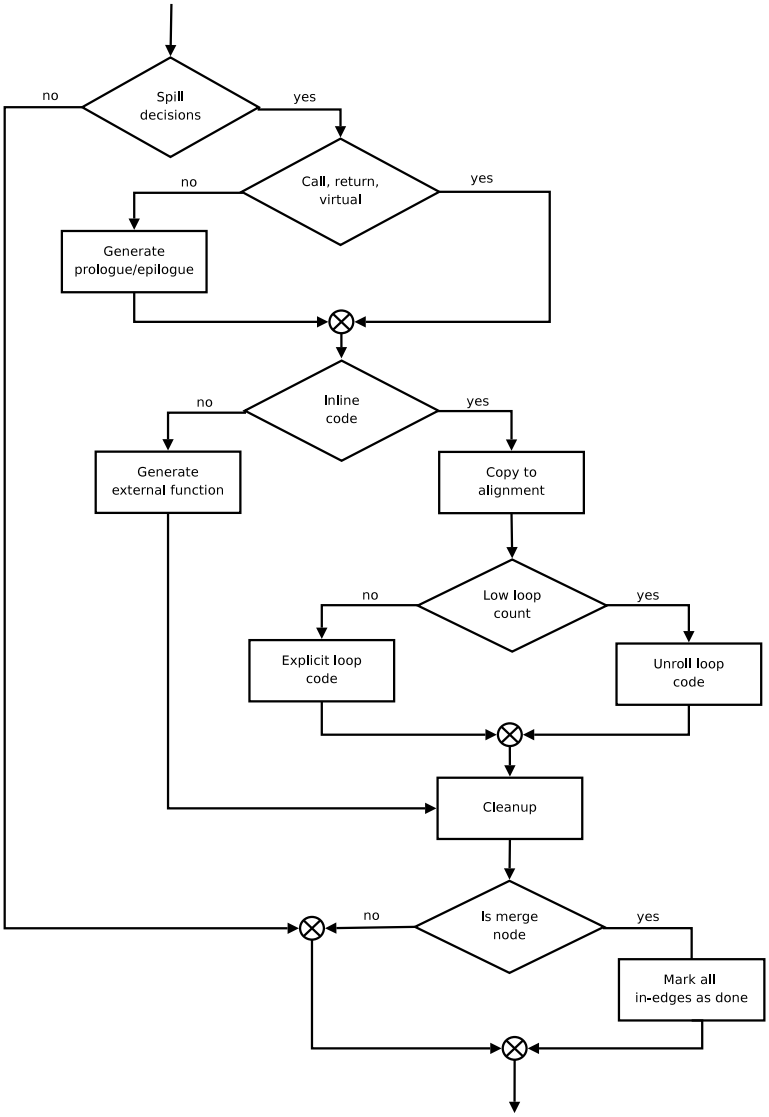


Figure 5.13: Workflow of spill code generation

be considered to cause no overhead anymore apart from the memory accesses. Therefore, if the estimated loop iteration counts are lower than three, the loop is unrolled. After having emitted the copy code, some basic blocks that have been generated by the different stages above can be merged. This is done in the cleanup step. If the target node of the edge is a merge node, all spill decisions on all in edges are marked as handled as an appropriate spill code has already been emitted. It should be noted that the prologue and epilogue code only backs up registers that are actually used. Therefore, the overall overhead is minimized.

Of course, the spill code has implications for the ILP. On the one hand, the spill code increases the program code size. On the other hand, the spill code must be considered in the structural model to properly obtain the basic block WCET. Spill code is never emitted into

the scratchpad memory, therefore the former is not a problem to the allocation. In the second case, an estimation of execution times must be provided to ensure the correctness of the ILP model. In equation 5.12, the spill cost function c_{copy} was introduced. The precise cost of the copying cannot be known within the ILP model because groups are not available. However, an overestimation is still possible. According to the workflow presented in figure 5.13, the accumulated costs can be roughly determined. For reading memory accesses from the program flash 5 cycles are assumed, while writing to the scratchpad memory is estimated with 1 cycle. Only the copying of single basic blocks can be considered within the ILP which leads to a safe overestimation. For each single basic block it must be assumed that it ends with an explicit jump instruction since the order of basic blocks in the target memory is unknown upfront.

The workflow chart already suggests how the spill costs are determined. Since the exact code shall not be discussed at this point, the following constants are defined to represent the costs of the separate steps during spill code generation:

- C_{save}: The costs of saving and restoring the registers required in the spill code
- C_{loop}: The costs of the inlined spill code including alignment and considering loop unrolling.

The spill costs as used in equation (5.15) can now be defined as:

$$c_{copy} = \begin{cases} 0 & \text{if spilling on the virtual entry edge of the IPCFG} \\ C_{loop} & \text{if spilling on a call or return edge} \\ C_{save} + C_{loop} & \text{otherwise} \end{cases} \tag{5.16}$$

If a spill decision was made for the virtual entry edge of the program which was introduced in definition 5.2.9, then no costs are applied at all. This results in a set of objects preloaded into the scratchpad memory prior to execution. This is similar to the static allocation. If spilling is required on an edge leading from or to a call site, then no explicit register saving and restoring is needed because no registers that could potentially carry arguments or return values will be used in the spill code. All other registers are potentially unused at these points. For all other edges, an explicit epilogue and prologue is required.

5.7 Implementation

The implementation of the dynamic allocation is a specialization of the static allocation. Much of the routines utilized there could be reused as far as the structural model and the jump correction is concerned. They are discussed in sections 4.5.1 and 4.5.2.

The workflow for solving the dynamic allocation problem is summarized in figure 5.14. Almost all steps have already been discussed. The input to the optimization is again the low-level representation LLIR. From this LLIR, intraprocedural control flow graphs are

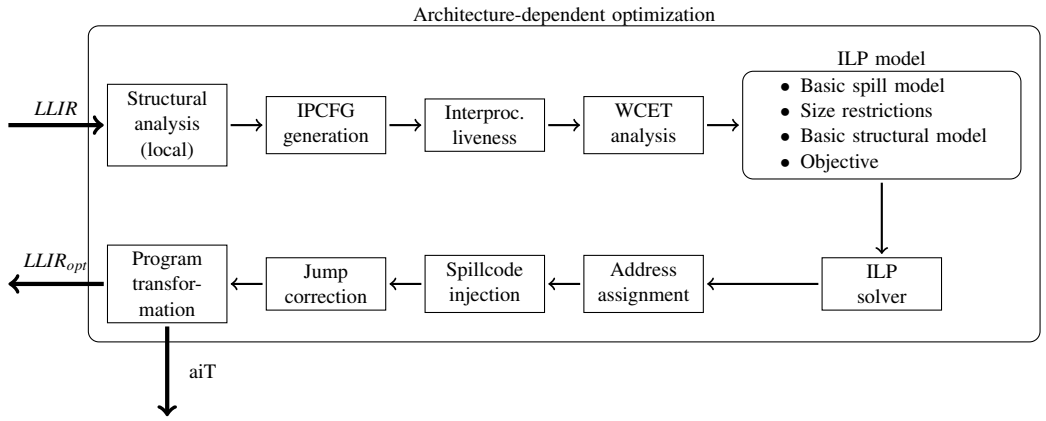


Figure 5.14: Workflow of dynamic optimization

constructed which will later serve as the source of structural and WCET-related information. The structural analysis (section 4.2) determines the nesting of constructs within the code so that this information is available during ILP generation. From the separate control flow graphs an interprocedural graph is constructed. With its help, the liveness of objects across function boundaries can be determined (see section 5.2). Afterwards, the WCET is determined for the unaltered program. This is again performed in two passes. The program is fully assigned to a memory on each pass. From the assembled information, the ILP model is constructed as was discussed in section 5.4. The spill model is reponsible for propagating allocation decisions. The size restrictions prevent the overuse of the scratchpad memory. With the basic structural model, ILP constraints similar to the ones in the static allocation problem are generated. The decisions from the spill model influence the estimated costs in the structural model. After the ILP has been solved, an address assignment (section 5.5.2) for the selected objects needs to be found. Afterwards, the program is altered so that the spilling of objects is performed in terms of the regular execution flow of the program (section 5.6). The placement of basic blocks into different locations also requires a correction of jump instructions (section 4.5.2).

The final step, referred to as *program transformation*, has not yet been addressed yet. The dynamic allocation hits a technical barrier when it comes to evaluation of the optimized program and the output of executable code. This is a fundamental restriction which is discussed in the next section.

5.8 Program transformation

The outcome of the dynamic allocation steps as described, are grouped basic blocks and memory addresses for their placement. Using this information an executable program has to be generated.

A key problem in this regard is the fact that objects occupy overlapping address intervals over the course of the execution time. This was illustrated in figure 5.10. The jump correc-

tion process and moreover the spilling routines need to be crafted for this scenario.

The jump correction performs its task by calculating the distance between the objects and by replacing instructions on demand. This is required if the distance between the objects is so large that the existing instructions are not sufficient to express a jump of that distance. This is mostly related to the limited widths of the immediate operands in the TriCore1 instruction set.

Similarly, the spill code requires information on the source and the target addresses as well. The problem of determining the target addresses has been solved by the algorithm presented in section 5.5.2.

The LLIR symbol table, as presented in section 3.4.2, is capable of managing object addresses. In particular, it enables the utilization of symbols instead of immediate addresses within the instruction code. The tables have been contributed to the LLIR to relieve the user from the explicit address management. However, in this situation, overlapping objects need to be managed. This poses two problems to the generation of valid programs.

The composition of the program object files requires a relocation by the linker. However, neither the model provided through the LLIR symbol tables nor the standard configuration of the linker are capable of handling overlapping objects. A special *OVERLAY* directive [Pro07] for linker scripts can enable this. The LLIR symbol tables however currently do not support overlay.

The only way to overlay objects for linking is to provide a separate object section (refer to section 3.4.2) per basic block group. The OVERLAY directive allows to specify the overlapping of sections within a memory. Since the basic block groups are determined over the course of the allocation, a manually prepared linker script cannot be used. The only way to encode the overlay is to dynamically generate a linker script from the results of the allocation with the help of the LLIR extensions implemented for this thesis. As a consequence, the symbol tables would need to be capable of calculating much more difficult scenarios to determine the final object addresses. The alternative would be to fully refrain from using symbolic addresses and to actually perform the relocation step within the compiler as well. This would be the equivalent of provide a fully featured linker as an integrated compilation step. In the context of this thesis, this is an unrealistic undertaking under the given time constraints. A fully instruction-corrected assembly output can be generated. But the description of overlaying objects is not available and as a result, the final linker invocation is not possible without providing the additional information as just discussed. This is denoted as the outgoing edge labelled $LLIR_{opt}$ in figure 5.14.

This restriction to the final program output does not apply to the aiT WCET analysis. A whole different issue becomes apparent at this point. As was presented in section 3.1, the aiT WCET analyzer operates on an intermediate representation (*CRL*). The *LLIRAIT* component of the WCC transforms its internal LLIR representation into the CRL. Numerous aspects of the source program need to be described in this representation. In particular, all objects are given a memory address, including single instructions. The input to aiT must be a complete and static model of the program. Moreover, no two instructions may be assigned overlapping addresses. It is therefore technically not possible to model a self-modifying program for WCET analysis.

As a consequence, the input provided to aiT must be a *flattened* model of the program. Objects that are assumed to be overlapping, according to the memory and address assignment, must be placed adjacent to each other, so that an appropriate model can be delivered to the analyzer. It means that a structure similar to that of the static allocation has to be provided. All objects must be statically placed and may not overlap. In turn, this resolves the problem that was addressed before. The fact that overlapping cannot be analyzed means that it also is not required to be supported by the LLIR extensions. A specially transformed input is provided to aiT as denoted by the the outgoing edge labelled *aiT* in figure 5.14.

However, the result of the flattening inevitably has an impact on the precision of the WCET analysis. The jump instruction correction now needs to address jump scenarios which suggest much larger address intervals than it would have been the case in reality. This affects the estimation of the WCET. In the follwowing chapter, the results will be investigated.

CHAPTER 6

RESULTS

6.1 Benchmarks

The allocations presented in the previous chapters need to be tested in a comparable fashion. The WCC comes with a large benchmark-suite specifically aimed at measuring results of WCET-directed optimizations. Most of them implement algorithms for DSP-related tasks. This is an important domain to embedded systems and standardized benchmarks assist in obtaining comparable results for optimizations. The algorithms in use are dedicated to image, video, audio encocoding and decoding. Other benchmarks are aimed at providing test cases for typical algorithmic problems like sorting or matrix transformations. The remaining benchmarks are related to simulating certain scenarios like the simulated execution of a petrinet or typical arithmetic operations like calculating the square root.

The benchmarks are generally self-contained. This means that no explicit dependencies to external library functions exist. This also implies that no benchmark imposes side-effects or asynchronous behavior. The input to the algorithms is usually provided as static data in the source code or by generator functions. Most benchmarks consist of single files. Moreover, it is attempted in general to avoid program structures that make the structural analysis of the code particularly difficult. These are statements like *break* and *goto*, and the use of multiple *return* statements per function. Pointers are not used in general. The dynamic behavior is minimized. Summing this up, the benchmarks in general completely fulfill the requirements for being good candidates for static code analysis as discussed in section 2.2.2.

These benchmarks are originally not prepared for their direct use in WCC. To make them applicable to the analysis by aiT WCET analyzer, explicit code annotations are provided. All loop constructs are annotated with the respective loop bounds.

The properties of the available benchmarks are listed in appendix B. As can be seen, they greatly vary in properties like total size, number of basic blocks, number of loops, etc. In general, they are small. The total size listed in the table only denotes the size of the resulting instruction codes at optimization level 2 of the WCC. General data is not considered. For the purpose of this thesis, some benchmarks which significantly differ in their properties are presented exemplary. The aim is to compare benchmarks with few large functions to ones with many small ones, to ones which incorporate comparably many loop constructs, etc. All benchmarks were conducted with an uncached main memory.

6.2 Methodology

As can be seen from the table of properties, the memory footprint of the instruction codes is comparably small. The scratchpad memory for instructions has a size of 48KB whereas the benchmarks have an average size of just 2.8KB.

The software emulation of the optional floating point unit on the TriCore1 architecture is disabled. This prevents library calls. Instead, floating point instructions are emitted which is fully supported by the aiT WCET analyzer.

For the purpose of benchmarking, the scratchpad memory size has been artificially decreased. Its size is determined relative to the program size. In the following discussion a memory size of 100% denotes a scratchpad just as large as the respective program. For the benchmarks, the sizes are varied in steps of 10% downwards. To make the allocation behavior comparable, the performance is denoted as the gain in percent compared to the execution from the main memory.

The results are obtained by running WCC with the respective compilation options. The WCET analyses are performed as an integral part of the compilation process. To test against different memory sizes, their size can either be specified in the external file which describes the memory layout as presented in section 3.4.2, or a command line option can be set. The latter has been temporarily added for benchmarking purposes.

In the following, benchmark results are presented that demonstrate the particular behavior of the static and the dynamic allocation optimizations, respectively.

In appendix A, the absolute cycle counts of the benchmarks are listed. In addition, a table of properties of the benchmarks like size, number of loops, number of basic blocks etc. is given in appendix B.

6.3 Results of static allocation

In the following, results of the static allocation optimization for code objects are presented.

The results shown in figure 6.1 stem from the benchmark *ndes*. It performs the DES encryption algorithm on static input data. The source code is composed of 12 loops and 5 functions. The loops are not nested. Primarily, it performs bit operations on the input data. If the scratchpad size matches the program size (100%), a gain of 39.2% can be achieved. The static allocation tends to select complete loop bodies. In this benchmark, the comparably large number of loops not being nested results in a gradual decrease of the gain. At 80% in size, a comparably expensive loop is dropped from the scratchpad memory which results in a gain of 24.1%. The increase to a gain of 26.5% at a size of 70% appears due to the imprecision of the model in comparison to the complexity of the architecture. For example, although the expected overheads due to different jump scenarios are taken into consideration, neither the state of the pipeline, nor the state of memory controller are taken into account. In this particular example, the result is an increase of gain of 2.4%, although

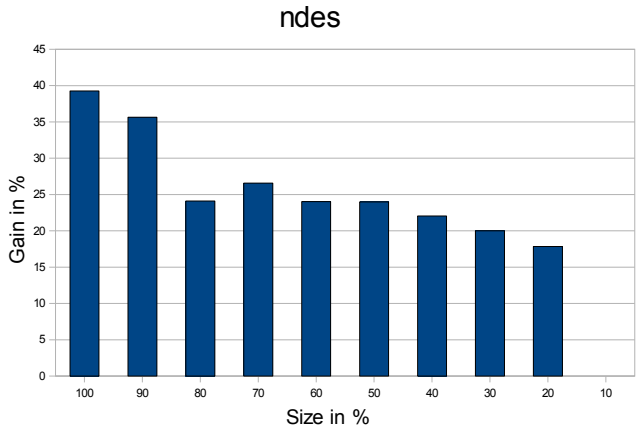


Figure 6.1: Static allocation results for benchmark *ndes*.

the size of the scratchpad memory was reduced. Otherwise, the gain decreases monotonically as expected. At 20%, a gain of 17.8% can still be achieved. At 10%, the allocation is not performed because no loop fits into the scratchpad in its entirety.

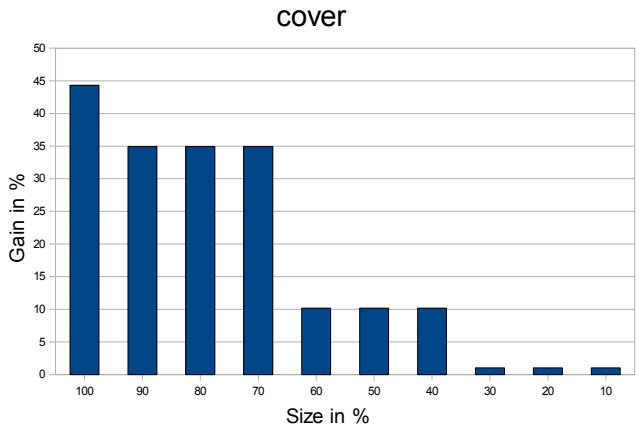


Figure 6.2: Static allocation results for benchmark *cover*.

Figure 6.2 illustrates the result for the *cover* benchmark. The source code is structually comparably extreme in that it encompasses 3 loops containing large switch statements. Its overall size is 3KB. Due to the very high count (399) of small basic blocks (10B max. size), the overhead imposed by partially allocating the loops to the scratchpad memory quickly exceeds the possibly achievable gain. The measuring points at 90%, 60% and 30% of the size suggest that one loop after another is kept in main memory. The result is that at 100%

a gain of 44.3%, at 90% a gain of 34.9%, at 60% a gain of 10.1% and at 30% a gain of 1.0% can be achieved. The gain at very low scratchpad sizes stems from the placement of the entry function which invokes 3 functions containing the loops, respectively.

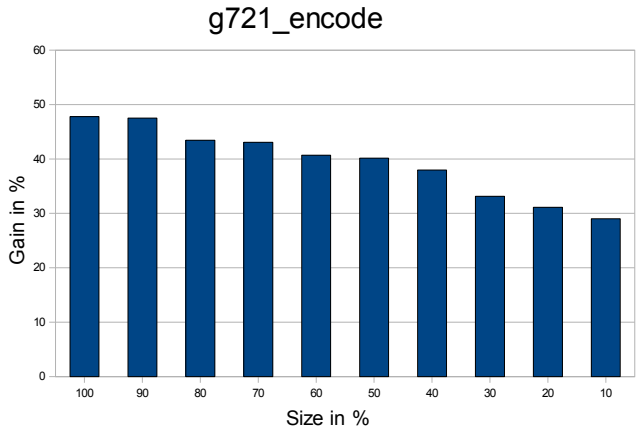


Figure 6.3: Static allocation results for benchmark *g721_encode*.

The results of a benchmark which implements the G.721 ADPCM audio encoder are shown in figure 6.3. The source code consists of 11 loops and 26 functions. Its total size is 3.1KB. Most functions consist of non-iterating sequences of basic blocks. As a result, the selection of objects for allocation into the scratchpad memory is gradual for scratchpad sizes of 100% to 10% of the program size. With a scratchpad size of 100% of the program size, a gain of 47.8% results. At 10% of the size, a gain of 29% can still be achieved. A small central loop with a maximal loop iteration count of 256 exists, which can be kept in the scratchpad memory even for very small sizes. This explains the comparably high gain at only 10% of the original program size.

Figure 6.4 shows the average gains that can be achieved with the static scratchpad allocation. The maximal achievable improvement for a scratchpad size of 100% of the program sizes is 39%. The largest benchmark is *md5* (6.3KB), the smallest *fibcall* (52B). The average code size among all benchmarks is 2.8KB. It can be seen that for scratchpad memory sizes as small as 10% of the original program sizes, an improvement of 7% can be achieved. The actual improvements per benchmark can differ largely, as was presented above.

The TriCore1 processor features an instruction cache of 16KB. For all the benchmarks, up to this point, the results are provided in reference to the gain in performance in comparison to an uncached main memory. Since none of the benchmarks has a code footprint greater than the actual cache size, the comparison of a cached memory with the results of a static allocation requires an artificial reduction of the cache memory size. In the following, the cache size is equal to that of the scratchpad memory.

The advantages of a cache are that it operates transparently to the processor. Therefore, no instruction modifications need to be applied to make use of the faster memory. Its disad-

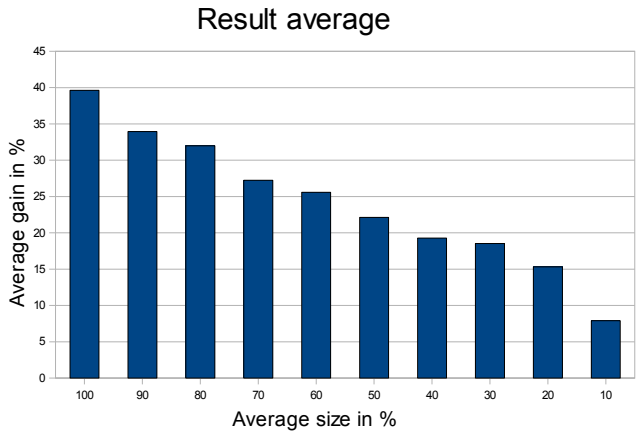


Figure 6.4: Static allocation average results

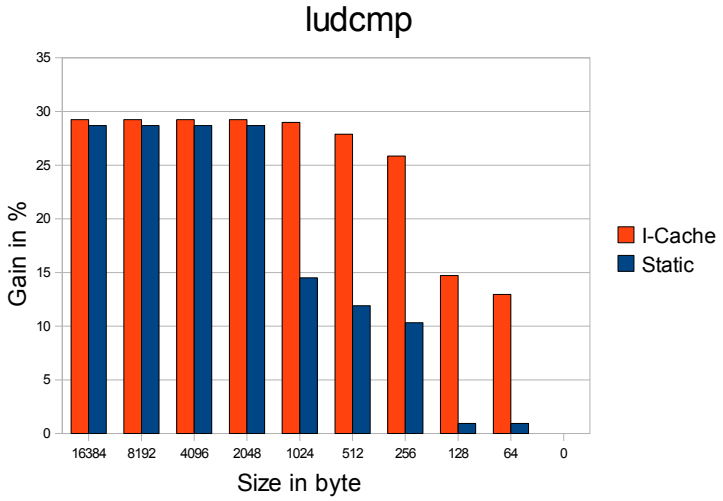


Figure 6.5: Static allocation compared to the I-Cache

vantage is the lack of control from the perspective of a user. The aiT WCET analyzer for TriCore1 provides a precise model of the instruction cache. A direct comparison shall be provided at this point. In figure 6.5, the results of the *ludcmp* benchmark are illustrated. It implements an algorithm for solving linear equations through LU decomposition. It has an instruction code memory footprint of 1.5KB and is consists of 11 loop structures which are mostly responsible for iterating the rows and columns of a matrix. As can be seen, for memory sizes larger than the program size, both results are almost identical with a gain of

29.2% for the cache and 28.7% for the scratchpad memory. From a size of 1024B to 256B, a significant drop in the gain for the static allocation optimization can be observed. The gain decreases to a gain of 14.5% for a size of 1024B , 11.9% for a size of 512B and 10.3% for a size of 2565B, respectively. For sizes smaller than 128B, the gain achievable by utilizing the scratchpad memory is minimal with a gain of 0.9%. Opposed to that, the use of the instruction cache results in sigificantly higher gains even for small memory sizes. For sizes of 1024B to 256B the results indicate a gain roughly twice as high as with the scratchpad optimization. Even for a size of 64B (this corresponds to 2 cache lines), a gain of 12.9% can be achieved. The results seem to suggest an advantage of the instruction cache. Clearly, an overhead results from the modifications necessary to execute the program utilizing both memories, when the scratchpad allocation is performed. However, the locality of execution is comparably low. Most of the 11 loops in the benchmark are executed in succession. The cache seems to be filled even prior to execution, or at least is considered filled immediately by the analysis. This problem regarding the cache simulation of the aiT WCET analyzer has also been observed in other contexts. For the current benchmark, even with the full program assigned to the scratchpad memory (scratchpad memory size is 100% of the program size), the use of the cached memory results in a higher gain of 0.5%. But for the cache to take effect, at least a single access to the relevant code objects in the main memory must have been performed whereas the program assigned to the scratchpad memory can already execute at its full speed. The maximal iteration count per loop is only 5. This indicates a problem with the analysis of the ait WCET analyzer. Therefore, the given results must be taken with a grain of salt and further, thorough comparisons of the scratchpad allocation techniques with the instruction cache seem unreasonable at this point in time.

In general, it should be noted that the sizes of the benchmarks are comparably small. The static optimization takes into account the dynamic growth that stems from the allocations. For the worst case, an overhead due to instruction corrections up to 12B needs to be taken into account per jump instruction. By enforcing such a pessimistic estimation a guarantee exists to not exceed the scratchpad memory size. On the other hand, a more precise estimation could possibly lead to even better results - specifically for small benchmarks where the necessary instruction modifications already make up a noticeable part of the overall memory footprint.

6.4 Results of dynamic allocation

In this section, the results of the dynamic allocation optimization for code objects are presented. Similar to the results of the static allocation, an assorted set of benchmarks with interesting properties is investigated.

The results for the *fdct* benchmark are shown in figure 6.6. The benchmark performs a Fast Descrete Cosine Transformation on static input data. The total size of the instructions is 1.8KB, with few but large basic blocks (260B average). The control flow consist of just 2 loops with a maximum iteration count of 8. According to this, the allocation results reflect the placement of the loop bodies into the scratchpad memory. For a full assignment (size of 100%), a gain of 14.5% can be achieved. At sizes of 90% and 40%, the loops are dropped

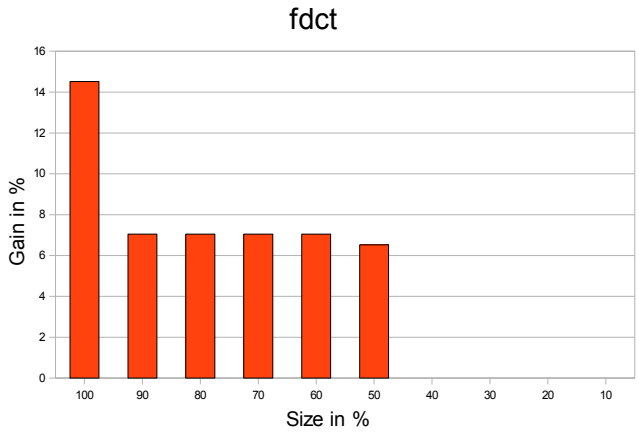


Figure 6.6: Dynamic allocation results for benchmark *fdct*.

from the scratchpad memory, respectively. This results in a gain of 7.0% for a size of 90% the program size to a gain of 6.5% for a size of 50% the program size. For smaller scratchpad memory sizes no allocations are performed. For the whole benchmark, no spill code is emitted at all. The overhead in terms of size and execution time do not justify this. As a result the results of the static and the dynamic allocation are equivalent.

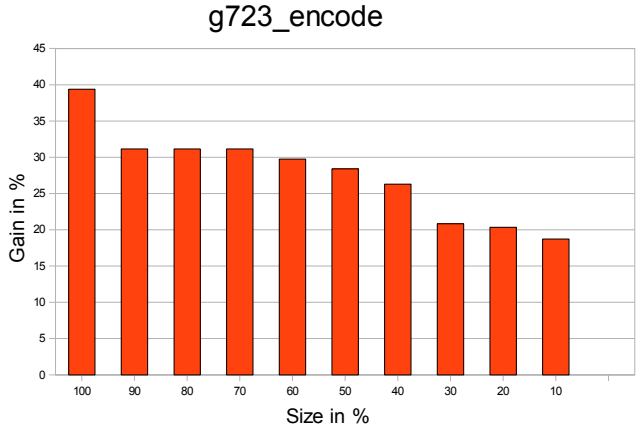


Figure 6.7: Dynamic allocation results for benchmark *g723_encode*.

The next benchmark, shown in figure 6.7, implements the G.723 ADPCM audio encoding algorithm. Its instruction code size is 3.1KB. It consists of 209 basic blocks with an average of 15B each. In total, 11 loop constructs make up the algorithm. A central loop in the entry function invokes the encoding functions. The maximum gain with a scratchpad size of 100%

the program size is 47.4%. Decreasing the scratchpad memory size to 90% results in a gain of 31.5%. This result does not change for sizes of 80% and 70%. This is due to many basic blocks not being part of loop bodies. Not allocating them to the scratchpad memory has no noticable effect on the result. From a size of 70% downwards, a gradual decline occurs where the small separate loops are not allocated. Due to the small sizes of the loops, at only 10% of the program size a gain of 18.7% can still be achieved.

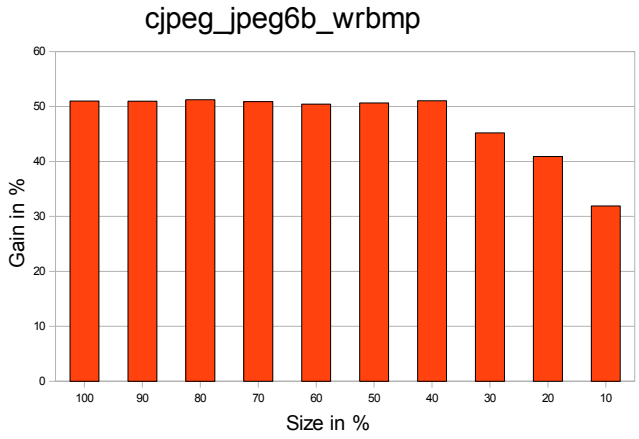


Figure 6.8: Dynamic allocation results for benchmark *cjpeg_jpeg6b_wrbmp*.

In figure 6.8, the result of the *cjpeg_jpeg6b_wrbmp* benchmark are shown. It generates an image file in the BMP format. It encompasses 55 basic blocks and has a total instruction code size of 638B. The control flow consists of 6 loop constructs with maximum loop bounds of 30 to 512 iterations. The majority of code is not nested within these loops. Because of this, the allocations result in a constant gain of 51% from 100% to 40% of the original code size. The loops are kept in the scratchpad memory as long as possible. From 30% to 10%, a significant decrease of gain can be observed. At 30% a gain of 45.1% and at 20% a gain of 40.8% can be observed, respectively. At 10% of the size, a gain of 30.8% is achieved.

The average improvements over the applied benchmarks are shown in figure 6.9. As can be seen, with a full allocation to the scratchpad memory (100% of program size), an average gain of 37.1% can be achieved as opposed to the execution from the main memory. At 90%, 60% and 30% of the size, the average gain decreases to 32%, 26.6% and 18.6%, respectively. At 10% of the original program size, an average gain of 9% can still be achieved. The decline is gradual as expected.

Not all of the benchmarks performed for the static allocation optimization could be successfully applied for the dynamic case. This has various reasons discussed in section 6.7.

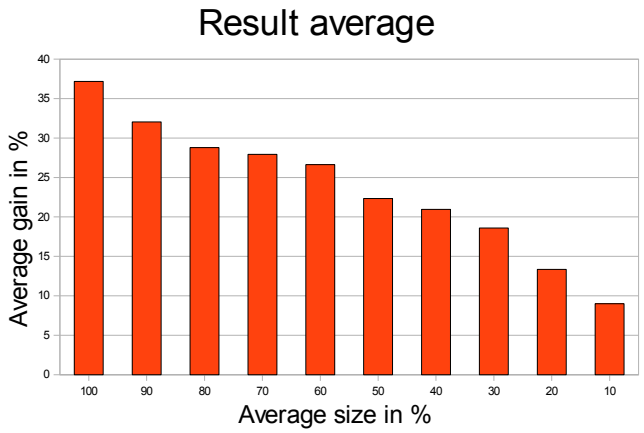


Figure 6.9: Dynamic allocation average results

6.5 Comparing static and dynamic allocation

In the following, a direct comparison of the static and the dynamic allocations is presented. In the dynamic allocation, the exchange of the contents of the scratchpad memory can lead to greater gains. For the benchmarks, this is especially true for control flows which are clearly separatable into stages of processing. Two such benchmarks are presented.

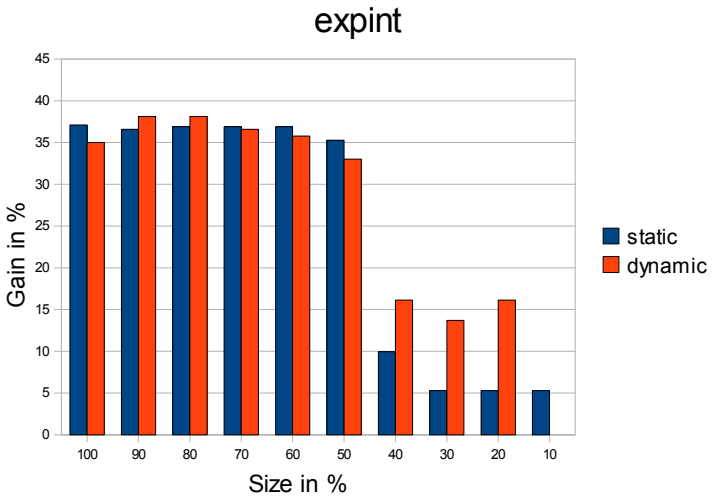


Figure 6.10: Comparison of result of benchmark *expint*.

Figure 6.10 illustrates the comparison of the *expint* benchmark. The algorithm calculates the exponential integral for fixed input arguments on a finite interval. The memory footprint is 800B large and is composed of 21 basic blocks with an average size of 38B. The implementation consists of 3 loops with a maximal loop bound 100 iterations. With a scratchpad size of 100% of the program size, the gain of the static allocation is 37%, whereas the gain for the dynamic allocation However, as can be seen, the static allocation results in slightly better results. This is due to the strictly more conservative estimation of growth for the dynamic allocation which also has to take into account a different order of basic blocks within the scratchpad memory. Therefore, no equivalent set of objects is chosen for allocation. At 90% and 80%, the different placement even seems to lead to increasingly higher WCET gain estimations in the case of the dynamic allocation. This issue is discussed in more detail in section 6.7. For scratchpad sizes smaller than 70% of the original instruction code size, the dynamic allocation in general performs better than the static allocation. As can be seen at from 50% to 40% size, a large drop in gain can be observed from 35.2% to 9.9% for the static allocation and from 33% to 16.1% for the dynamic allocation. This is due to an expensive loop construct being dropped from allocation. Down to a size of 20%, the dynamic allocation results in a gain of 16.2% whereas with the static allocation a gain of only 5.2% can be achieved. At 10% of the original size (80B), the static allocation can still achieve a gain of 5.2% whereas the dynamic allocation leads to no allocations anymore. The latter observation already points out a weakness of the dynamic allocation optimization. In section 6.7, these problems are addressed.

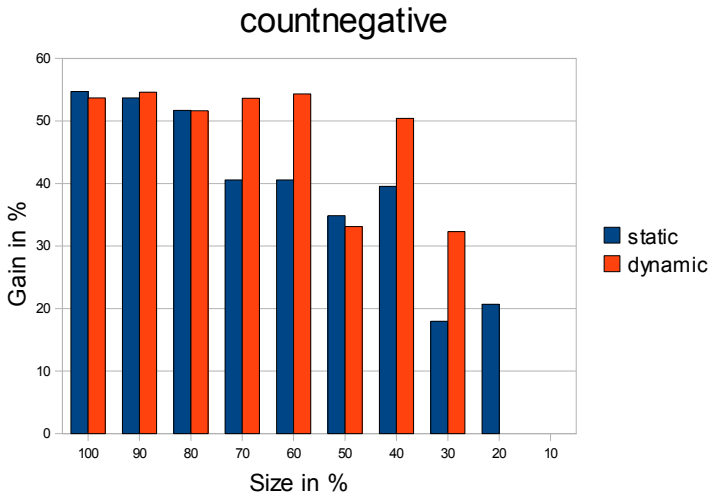


Figure 6.11: Comparison of result of benchmark *countnegative*.

The benchmark whose results are shown in figure 6.11 implements an algorithm which works in two successive stages. A square matrix of 400 elements is filled with pseudo-random values. Afterwards, the number of non-negative values is counted. The benchmark

consists of 18 basic blocks which accumulate to a total instruction size of 260B. With a scratchpad memory size of 100% to 80%, both allocation techniques result in almost identical gains. The slight differences stem from different requirements regarding the estimation of the dynamic code size inflation. The maximum gain for a size of 100% is 54.7% for the static allocation and 53.7% for the dynamic allocation. At sizes of 70%, 60%, 40% and 30%, the spilling leads to significant improvements in comparison to the static allocation. At 70% and 60%, a gain of 40.6% for the static allocation and a gain of 53.6% and 54% for dynamic allocation can be achieved. At 40%, gain of 39.5% as opposed to 50.4% can be achieved. At 30%, a gain of 17.9% as opposed to 32.3% results. However, at small sizes (20%), no placement is performed at all for the dynamic allocation anymore. The gain of the static allocation is 20.6% for this size. At 40% size, it can be observed that the gain according to the WCET estimation for the dynamic allocation is greater than at 50% of size. This issue is also addressed in section 6.7 below.

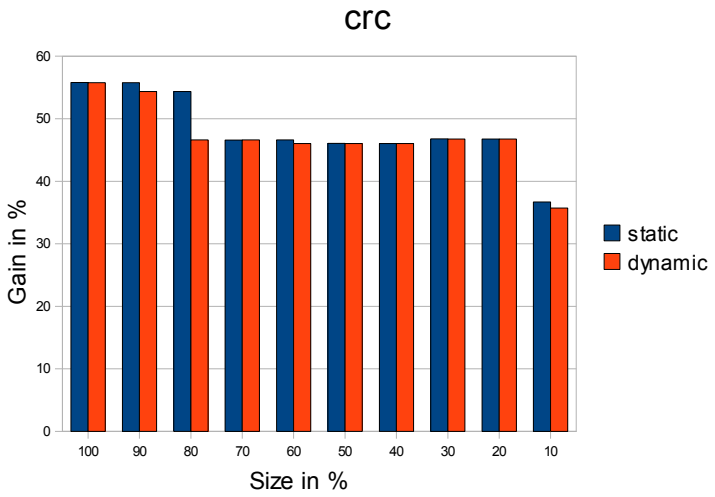


Figure 6.12: Comparison of result of benchmark *crc*.

Figure 6.12 illustrates the comparison of the *crc* benchmark. It is an example of a closed algorithm which performs its primary processing within a single tight loop. Its total instruction size is 608B. As can be seen, the gains resulting from different memory sizes are almost equivalent. For a scratchpad memory size of 100% a gain of 55.7% can be achieved. At a scratchpad memory size of 80%, the more conservative size estimation leads to a difference in allocation, so that the gain for the static allocation is 54.3% and for the dynamic allocation 46.6%. From 70% of the program size (gain of 46.6% for both allocations), the gains remain equivalent. At 10%, a gain of 35.7% is still possible. The benchmark demonstrates the equivalence of results due to the equivalence of the structural model. No decisions to spill are made for this benchmark.

For many benchmarks, the results of the static allocation do not differ significantly from

those of the dynamic allocation. The reason for this is the comparably high overhead of the spill code. As was discussed in section 5.6, the spill code can quickly grow in size. Even for the transfer of just a single object, instructions for saving the context and restoring need to be provided, so that registers can be freely used in the actual spill code. Besides context saving, which can have a size of up to 40B alone, another 40B are required for setting up the spill code itself. The average size of a basic block in the benchmarks is only 32B. In the regard of execution overhead and size overhead, this is significant. In many cases, a static set is chosen to be loaded prior to execution which remains in the scratchpad memory. Since the spilling is bound to object live ranges and many benchmarks perform a single closed algorithm, exchanging the contents of the scratchpad memory rarely occurs.

A direct comparison of the average results of both allocation techniques reveals that at 100% of the program size, the static allocation results in an average gain of 39.6% as opposed to 37.1% for the dynamic allocation. At 50% of the size, both allocations result in an average gain of 22.1% for the static allocation and 22.3% fir the dynamic allocation. For a scratchpad memory size of only 10% of the original program size the static allocation results in an average gain of 7% whereas the dynamic allocation results in a gain of 9%. Both allocations lead to a gradual decrease of gains. These observations validate the expectations for both allocation optimizations. For comparably large sizes of the scratchpad memory, the static allocation performs better. This is due to the lower overhead caused by instruction corrections. On the other hand, for small scratchpad memory sizes, the dynamic allocation can achieve better results because the scratchpad memory contents can be exchanged at execution time.

6.6 General limitations

In this section, the problems encountered during implementation of both the optimization techniques are discussed. Despite the correctness of the ILP models, several technical limitations became apparent.

Both optimizations do not support recursion in general. This is due to two reasons. For both models, the iteration counts of loops or the depth of recursions must be statically known to reason about the costs induced by the elements of a program. Direct recursions can in fact be handled easily as far as these iteration counts are concerned. If a function calls itself, the costs of the recursively executed paths can be obtained directly. For indirect recursions, matters are more difficult. Multiple invocations can lead back to one originating function. As was discussed in section 3.1, possible manual code annotations can be loop bounds that serves as the source of information for language statements directly following the annotation, or they can consist of markers and loop bound equations which describe the relation of execution times of the markers. Markers can be arbitrarily distributed within the source code. Although they are a means to describe complex iteration scenarios like recursions, an additional effort would be required to properly take into account the equations within the respective ILP. Also, recursions lead to infinitly long call strings during the interprocedural liveness analysis. Since only very few of the benchmarks are recursive, the author decided to drop the explicit support. An experimental implementation is provided for the static allo-

cation optimization as a proof of concept. It revealed an unreasonable additional effort for fully supporting recursion in this thesis.

A weakness of both optimizations is the dependence on the structural analysis presented in section 4.2.2. It makes the generation of the ILP significantly more convenient, because the generated control tree directly reflects the program structure in the way it is required for the ILP generation. The drawback that became apparent for some realworld test cases is its fragility. Due to the compiler optimizations performed or not performed prior to the allocation optimizations, the program structure can cause problems. It was observed that the analysis of the very same program with only slight modifications due to some other optimization can lead to different results. This is because the analysis can possibly detect structures differently depending on the order of traversal. It was observed that especially nested structures with *natural loop* statements can cause problems, because they have no dedicated basic block denoting the entry into the loop. Structures therefore become indistinguishable. It should be noted that the optimization level `-O2` for the WCC leads to viable results in general. Problems with single benchmarks could be resolved by hand-tuning the optimization settings or by minimal changes to the source code. No generally working strategy for the structural analysis could be found. This is not a deficiency of the implementation per se. WCET analyzers require some form of a structural analysis as well. For example some are tuned to typical patterns produced by GCC [WEE+07]. The advantages of a detailed structural analysis are nevertheless indisputible.

In this regard, it should be pointed out that irreducible (refer to section 4.2.2) structures can in general be handled by the structural analysis. In the literature, multiple different techniques are proposed to deal with such situations. Only a single benchmark has been observed to be irreducible. Therefore, none of the techniques justified the effort of implementation. Optionally, the handling can still be implemented.

The static allocation optimization is in fully working condition and can be applied to arbitrary benchmarks in general. Opposed to that, several limitations apply to the dynamic allocation optimization. The specific problems are addressed in the next section.

6.7 Specific limitations of the dynamic allocation

As was already pointed out in the previous section, the spill code can significantly inflate the total program size and can also significantly contribute to an increase of execution time.

An important issue regarding the modeling of the allocation problems is the inflation of the program size in general. Not only does the spill code contribute to this, also the instruction corrections can have a large impact (up to 12B inflation per basic block). Because of this, a central aspect of both the allocations has been to dynamically model the inflation. The model proved viable for the static allocation to strictly prevent the generation of programs that do not fit into the scratchpad memory anymore. But it is critical to the dynamic allocation. Since the final block sizes are neither known while solving the ILP nor at the address assignment step, a suboptimal placement of code objects in the scratchpad memory occurs. The result is an inherent fragmentation which leads to more instruction corrections which

have a negative impact on the performance. Even for equivalent allocation decisions of both optimizations, small differences of 0.5% to 2% in the gain can be observed because of this. To the best of the author's knowledge, approaches to model dynamic size inflation have not been proposed in other works although this is a critical aspect to the utility of the scratchpad memory.

The problem just discussed can potentially be addressed. But the modeling of the impact of certain instructions and their placement on the overall performance requires a more detailed machine model. The TriCore1 has a comparably complex architecture. For the pipeline, the alignment of jump targets and the schedule of instructions are important. Moreover, the memory architecture would require a more precise model to better reflect the effects of memory transfers. Especially for small benchmarks, the estimations seem too imprecise. These effects occur mostly for very small (smaller than 100B) scratchpad sizes. On the other hand, for large benchmarks this is hardly observable. In part, this is related to the fact that scratchpad memory sizes are scaled according to the benchmark sizes for these evaluations. Concluding, experiments to the solution of these issues could not be conducted due to the limited time constraints of this thesis.

A very important aspect to the evaluation of the results is that technically, no realistic program could be generated. The aiT analyzer neither allows for overlapping code objects nor a writing access to the program scratchpad. It is not aimed at analyzing dynamic program behavior like this is the case for the self-modifying code in the dynamic allocations. The former problem was solved by generating a program representation where each object is placed in a non-overlapping fashion. In turn, this requires a lot more instruction corrections than was modeled in the ILP. Secondly, the actual transfer of code objects could not be performed because the code memory may not be accessed. Instead, a buffer in the data SRAM is allocated so as to at least perform a transfer. The costs of these transfers are much higher (up to 16 cycles as opposed to up to 6 cycles) as modeled in the ILP. In general, the dynamic allocation optimization performs as expected. The evaluated results, however, suggest a higher WCET than it would be the case with realistic input into aiT.

In conclusion, the primary problems stem from the complexity of the target architecture which makes it hard to provide adequate machine models. Moreover, the complex instruction set architecture makes the construction of a technical infrastructure inherently time consuming. The dynamic allocation, as far as the ILP model is concerned, shows an observably good behavior but lacks precision. In addition, the results from the WCET analysis can only be taken as a hint of the true results of the allocations due to the technical limitations just described.

CHAPTER 7

SUMMARY AND FUTURE WORK

In the following two sections, conclusions to the techniques presented in this thesis are drawn. In 7.1, the key issues addressed are summarized. In 7.2, propositions for possible extensions in the future are made.

7.1 Summary

The goal of this thesis was to propose WCET-directed allocation techniques for code objects. Two allocation schemes have been presented. On the one hand, a static allocation has been discussed where a set of objects is determined to be loaded into the scratchpad memory prior to execution of a program and which occupies this memory for the whole duration of execution. On the other hand, a dynamic allocation technique has been presented which determines the dynamic exchange of contents in the scratchpad memory during execution. Both techniques have been implemented as integrated optimizations of the WCC compiler.

The advantage of the allocation to a scratchpad memory as opposed to cache memories is the ability to fully control its contents. In the domain of embedded systems under hard real-time constraints, the usage of scratchpad memories is therefore a popular approach. The experiences gained from other works regarding energy reduction and WCET-directed optimizations have been utilized. However, for WCET-directed code allocations, only a single proposition exists which is based on the explicit enumeration of WCEP. In this thesis, techniques for efficient and integrated solving of this problem have been addressed for the first time.

For both allocations presented, a basic ILP model has been presented which solves the allocation problems by taking the WCEP into account only implicitly. Also, it addresses different technical constraints in addition to this. Besides the dynamic switching of WCEP, the dynamic size inflation resulting from allocation decisions has been taken into account. To the best of the author's knowledge, this problem has not been addressed in any other work related to scratchpad memory usage. Another particular problem during the allocation of program code is the optimal selection of groups of basic blocks. While for energy-related optimizations, different techniques for such a selection have been proposed, these proved

unsuitable for WCET-directed allocations. Therefore, a new technique for the dynamic selection of basic block groups has been proposed.

While the static allocation problem can be solved entirely within one ILP model, the dynamic allocation is conducted in two steps. During the optimization, an ILP delivers the mere allocation decisions and a post-processing step determines the actual placement into the scratchpad memory.

For both optimizations, significant work has been dedicated to the construction of a suitable framework. The WCC compiler was extended so that models of the memory hierarchy of the target machine can be used in the integrated optimizations. The framework which has been constructed is not limited to the allocation optimizations presented in this thesis. In particular, it allows a convenient access to information regarding the final physical layout of the program. This is a requirement for all optimizations taking multiple memories into account. In addition, this finished the integration of the aiT WCET analyzer into the WCC, as now arbitrary memory models can be provided. Also, an instruction-correcting compilation stage has been developed. Any assignment of code objects among existing memories potentially draws the program invalid as no suitable instructions have been generated in the first place by the general purpose code-selector. A correcting step restores the legality by performing optimal instruction replacements invisible to the user. These extensions are already actively used in other optimizations. In addition to this, an interprocedural data-flow algorithm has been proposed which serves the dynamic allocation by providing liveness information on code objects. Also, a structural analysis algorithm has been implemented which allows the reconstruction of a high-level program representation from a low-level, instruction-based input.

Both optimizations show good results during benchmarking. The static allocation model in particular could be shown to be good. A maximum average gain of over 39% could be achieved. In the case of the dynamic allocation, some technical limitations remain to exist. On the one hand, the dynamic placement of code objects could be improved by taking into account the state of the target machine for the allocations. The maximum average gain of 37% is lower than in the case of the static allocation but at small scratchpad sizes the allocation can achieve better results. At average sizes of 10% of the original program size, an average gain of 9% results as opposed to 7% for the average static allocation. The current results indicate that better estimations for the dynamic allocation can lead to better results. In particular, this is true for the generation of spill code. Due to the time constraints for this thesis, the implementation of the dynamic allocation could not be improved any further. On the other hand, the self-modification of the program during execution poses a limitation to the WCET analysis. Due to the technical restrictions of the aiT WCET analyzer, only approximate results of the dynamic optimization could be presented.

In conclusion, the thesis proposes novel ways for solving WCET-directed static and dynamic allocation problems for program code using scratchpad memories. Issues that have been largely ignored in the propositions to other, similar optimization techniques have been discussed and solved as far as technical limitations allowed this.

Referring to section 1.2, the goals of the thesis have been achieved. Moreover, many aspects that have not been considered beforehand needed to be taken into account. One of the

outcomes is a framework for optimizations related to memory hierarchies which completely replaced existing mechanisms and which is already used independently of this thesis in the WCC.

7.2 Future work

The optimizations presented in this thesis can still be improved in various aspects.

A possible future work is the support for recursive programs. The support for recursions in the optimizations discussed in this thesis has been dropped due to time constraints and because it is expected that recursions in general do not appear. Numerous other problems arise which need to be addressed. The correct modeling of iteration counts from flow facts is a challenging task. Also, the current approach of the interprocedural liveness analysis is not suitable for recursions as infinitely long call strings result. This issue has been addressed in other works and can be incorporated.

The structural analysis should be extended so as to support the full range of patterns of the WCC code selector. In particular, the problem of ambiguous loop headers needs to be addressed. Also, the handling of irreducible regions could be added. This is also an issue which has been addressed by other authors.

The static allocation model leaves little room for improvements. Opposed to that, the dynamic allocation could be improved in different technical and theoretical aspects. An important point is the lose coupling of the optimization stages. Especially the address assignment can be improved. In the current approach, a first fit allocation is performed. However, it showed that the simple placement, without taking effects on the execution behavior into account, leads to imprecisions. The integration of the address assignment problem into the ILP model is possible with the restriction that the determination of WCEP-bound basic block groups needs to be done in a different fashion than it was presented here. Also, this greatly increases the complexity of the ILP model. It remains the trade-off between precision and performance. The precise modeling of the target machine, especially regarding the instruction placement and the effects of code modifications, also remains a future work. In the current implementation of the dynamic allocation, the spill code is strictly placed into the main memory. It is possible to model its dynamic costs and its effect on the dynamic growth of the program with a more integrated model.

On the technical side, the infrastructure for modeling memory hierarchies could be extended to provide support for overlapping objects. Such an extension is still limited by the fact that the aiT WCET analyzer does not support self-modifying code. In the future, safer estimates from the dynamic allocation optimization should be possible if the technical prerequisites are fulfilled.

APPENDIX A

BENCHMARK RESULTS

Table A.1: Results of static allocation

Name	100%	90%	80%	70%	60%	50%	40%	30%	20%	10%	0%
adpcm_decoder	304974	305975	306205	306471	306648	306712	306865	288628	289032	289666	643709
adpcm_encoder	294922	295665	295961	296440	297351	290387	297282	297282	290896	291586	640787
adpcm_g721_board_test	177658	193898	188557	191579	196820	203817	209727	230052	253748	250515	302777
adpcm_g721_verify	183570	200311	195300	197845	204649	214345	217260	244112	262223	259547	312737
binarysearch	127	158	253	253	253	253	253	253	253	253	290
bsort100	111112	116092	111146	116014	107700	197620	197620	197637	202508	199238	199238
cjpeg_jpeg6b_wrbmp	148474	145704	147730	145714	145714	147856	146325	146325	174760	234692	298737
complex_multiply/float	122	145	145	181	181	181	181	181	181	181	181
complex_multiply/fixed	81	106	106	142	142	142	142	142	142	142	142
complex_update/float	320	354	354	354	402	402	402	402	402	402	402
complex_update/fixed	123	144	144	183	183	183	183	183	183	183	183
compressdata	1433	1456	1456	1431	1431	1525	1563	1563	1661	2435	2435
convolution/float	438	460	460	460	460	461	624	624	657	918	1022
convolution/fixed	539	540	540	540	569	569	569	569	844	844	840
countnegative	19254	19682	20531	25262	25262	27692	25703	34877	33712	42509	42509
cover	17566	20529	20529	20529	28346	28346	28346	31229	31229	31229	31552
crc	70358	70381	72621	84989	84935	85824	85853	84697	84734	100763	159120
dot_product/fixed	70	96	96	96	96	112	112	112	133	133	133
dot_product/float	141	169	169	173	179	198	215	215	215	215	215
edn	105391	105390	106239	106239	117166	119109	107563	107563	110248	112235	165970
expint	9015	9089	9046	9046	9046	9276	12909	13576	13576	13576	14333
fac	499	513	600	600	600	600	864	1027	1027	1027	1027
fdct	5009	5460	5460	5460	5460	5491	5874	5874	5874	5874	5874
fft_1024_13	154046013	152460504	154071711	155691154	157290126	154280366	152755134	168523300	168588929	168630807	169109948
fft_1024_7	152210482	152202416	152236176	155426902	152275138	152424463	152457137	168353325	168328720	168368593	168465967
fft_16_7	7050	18734	18984	7913	20421	8569	20951	8732	9036	22923	11117
fft1	37819	59623	83560	61145	61145	61145	62155	39759	87755	89007	54328
fibcall	278	290	290	336	336	336	723	723	723	723	723
fir/fixed	840	857	874	874	1030	1030	1032	1032	1062	1520	1650
fir/mrtc	661	679	668	668	867	867	870	870	870	1499	1499
fir/float	7749	7781	7784	11726	11726	11726	11726	11726	11726	11113	11113
fir2dim/fixed	6550	6657	7560	9079	9079	9079	9079	9079	10306	10164	10044
fir2dim/float	7301	7357	8072	9339	9339	9339	10248	9885	11042	10054	10640
g721_encode	797900	802164	864442	870170	906496	914847	948218	1022122	1053098	1085090	1528480
g723_encode	797100	804405	867471	885130	910784	925393	949163	1023293	1050206	1081154	1516268
h263	7190645	7233837	7377817	7377817	7368008	7708526	7708526	7725708	8381939	8381939	8310663
h264dec_ldecode_macroblock	114582	114768	141297	141297	141297	141297	141297	141297	141297	141297	142215
hamming_window	30186	30525	30525	29791	39800	39800	39800	39800	39800	39800	39800
iir_biquad_N_sections/fixed	720	739	775	775	775	775	775	781	1530	1520	1516
iir_biquad_N_sections/float	840	861	866	923	923	923	923	923	921	908	1116
iir_biquad_one_section/fixed	64	103	103	103	103	103	103	109	144	144	144
iir_biquad_one_section/float	87	113	113	113	113	113	113	116	133	133	170
insertsort	1193	1258	1258	1258	2164	2164	2164	2164	2164	2164	2164
janne_complex	284	295	303	736	736	736	736	736	736	736	736
jfdctint	5072	5170	5414	5414	5414	6380	6723	5814	5814	5814	7183
lcdnum	306	713	713	713	954	814	954	521	521	752	752
lms/fixed	1143	1155	1155	1155	1334	1334	1547	1548	1548	1571	1985
lms/mrtc	816227	820074	826121	848629	854751	871224	925268	968077	1143896	1206262	1344142
lms/float	1223	1251	1251	1274	1560	1543	1693	1701	1701	1866	2050
ludcmp	7268	8713	8713	8713	8713	8713	8749	8977	9181	10000	10190
matmult	801797	792642	793067	807042	808737	817542	913022	905321	891027	905857	900227
matrix1/float	23844	22167	21911	22008	35898	35898	35898	33162	35745	35305	41283
matrix1/fixed	25101	26214	25030	28636	40303	40303	40303	40303	43745	39359	39359
matrix1x3/float	386	410	412	502	502	502	502	674	660	756	756
matrix1x3/fixed	195	211	211	211	293	293	293	293	293	293	293

Table A.1: (continued)

Name	100%	90%	80%	70%	60%	50%	40%	30%	20%	10%	0%
matrix2/fixed	30844	30179	30392	43193	43193	43193	43193	43193	45408	43158	47253
matrix2/float	32957	33592	37727	42822	42822	42822	42822	42822	43036	48780	48780
md5	91894068	93345172	95047988	95047988	95047988	95047988	95047988	95047988	95047988	96462139	163159233
minver	4826	4866	5006	5626	5626	5626	5915	6185	6226	6626	7150
n_complex_updates/float	3644	3678	3678	3944	3944	3944	3947	4255	4255	4375	4375
n_complex_updates/fixed	3033	3062	3062	3308	3308	3308	3312	3312	3538	4221	4226
n_real_updates/fixed	1230	1263	1263	1263	1417	1428	1428	1428	1572	1912	1912
n_real_updates/float	1303	1336	1336	1336	1492	1504	1504	1504	1641	1949	1949
ndes	95307	100956	119079	115195	119183	119245	122303	125500	128887	156892	156892
petrinet	3474	5695	5695	5695	5695	5695	5695	5695	5695	5695	5695
prime	12155	12193	12193	12202	12218	20930	21691	25989	19995	29482	29482
qurt	4517	4545	4621	4621	4697	4697	4720	4727	4727	7751	7669
real_update/float	101	126	126	154	154	154	154	154	154	154	154
real_update/fixed	42	88	88	88	88	88	88	88	88	88	88
recursion	574	586	1611	1611	1611	1645	1645	1375	1375	1375	1375
searchmultiarray	16974	32880	32880	32880	32880	32880	32880	32880	32880	32880	32903
selection_sort	2064017	2153436	2331954	5022053	5022053	5022053	5022053	5022053	5022053	5022053	5022053
sqrt	8317	8334	8564	8564	8466	8466	10348	13602	13752	13752	13730
st	287719	287786	287769	297883	289772	323859	354814	323939	374844	463868	504767
test3	132871677	149934757	149398854	149398854	14364460	149547098	150034808	150811788	152468096	159659287	219872558

Table A.2: Results of dynamic allocation

Name	100%	90%	80%	70%	60%	50%	40%	30%	20%	10%	0%
adpcm_decoder	305193	309190	308730	311305	310908	310908	310005	309721	310697	292012	643709
adpcm_encoder	301116	301545	301867	303406	307157	307157	302291	302291	301253	301664	640787
binarysearch	151	158	290	290	290	290	290	290	290	290	290
bsort100	202851	210583	210583	217524	319337	319337	319337	319337	319337	319337	319337
cjpeg_jpeg6b_wrbmp	146432	228365	146719	146715	156320	158618	193816	175112	169038	203460	298737
complex_multiply/float	133	181	181	181	181	181	181	181	181	181	181
complex_multiply/fixed	92	142	142	142	142	142	142	142	142	142	142
complex_update/fixed	133	183	183	183	183	183	183	183	183	183	183
complex_update/float	358	358	358	358	358	402	402	402	402	402	402
compressdata	1542	2343	2460	1513	1475	2379	1563	1563	1661	2435	2435
convolution/fixed	584	759	460	460	533	528	819	815	918	918	1022
convolution/float	603	569	569	569	569	569	569	924	844	844	840
countnegative	19682	19300	20565	19707	19411	28439	21077	28772	42509	42509	42509
crc	70381	72621	84935	84935	85853	85853	85853	84734	84734	102310	159120
dot_product/fixed	94	94	94	94	112	112	112	106	133	133	133
dot_product/float	166	166	166	179	179	215	215	215	215	215	215
edn	76463	76463	76579	76579	76579	107627	107258	79267	109402	108959	135459
expint	9315	8869	8869	9089	9206	9601	12022	12370	12022	14333	14333
fdct	5021	5460	5460	5460	5460	5491	5874	5874	5874	5874	5874
fibcall	290	290	336	336	723	723	723	723	723	723	723
fir/mrtc	7785	7785	7784	11113	11113	11113	11113	11113	11113	11113	11113
fir/fixed	782	782	867	867	867	870	870	870	851	1499	1499
fir/float	857	857	1030	1030	1030	1032	1032	1062	1062	1520	1650
g721_encode	1061452	1086098	1086098	1086098	1132646	1159807	1188160	1236413	1236972	1287137	1528480
g723_encode	918976	1043975	1043975	1043975	1065241	1085646	1117705	1200543	1207715	1232306	1516268
h264dec…_block	112690	324807	434988	324916	324916	325431	153058	168852	328507	168009	151662
h264dec…_macroblock	123069	123216	149645	149645	149645	149645	149645	149645	149645	149645	142215
hamming_window	30525	30525	29791	29960	39800	39800	39800	39800	39800	39800	39800
iir_biquad_N_sections/float	739	775	775	775	775	775	770	850	1516	1516	1516
iir_biquad_N_sections/fixed	861	923	923	923	923	923	923	923	921	903	1116
iir_biquad_one_section/float	101	103	113	113	113	113	113	116	133	133	170
iir_biquad_one_section/fixed	77	79	103	103	103	103	103	109	144	144	144
insertsort	1308	1258	1258	1258	1480	2164	2164	2164	2164	2164	2164
janne_complex	295	303	736	736	736	736	736	736	736	736	736
jfdctint	5091	6054	6380	6380	6380	6380	6723	7183	7183	7183	7183
lms/mrtc	833143	898310	870414	848278	955015	1051788	1096267	1172952	1202893	1324223	1344142
lms/float	1348	1999	1999	1543	1543	1543	1693	1701	1701	2086	2050
lms/fixed	1219	1817	1817	1334	1334	1334	1547	1548	1548	1823	1985
ludcmp	7430	40864	8684	8684	8667	8828	9152	9152	9152	10001	10191
matrix1/fixed	26854	26109	25527	40303	40303	40303	40303	38833	38783	39359	39359
matrix1float	22257	22500	22500	35898	35898	35898	35898	33162	35745	36683	41283
matrix1x3/float	410	439	433	593	597	749	522	747	718	756	756
matrix1x3/fixed	211	211	211	295	293	293	293	293	293	293	293
matrix2/float	34361	34940	34957	42822	42822	42822	42822	42822	46264	48780	48780
matrix2/fixed	31811	31837	31153	43193	43193	43193	43193	43193	42373	45740	47253
md5	145086087	149647725	149647725	149647725	149647725	149647725	149647725	149647725	149647725	148719103	163159233
minver	6643	6763	7576	7576	7590	7730	8419	8483	12400	8818	7150
n_complex_updates/fixed	3299	3299	3299	3308	3308	3308	3312	3312	4094	4761	4226
n_complex_updates/float	3935	3935	3944	3944	3944	3947	3947	4811	4811	4375	4375
n_real_updates/float	1355	1355	1355	1355	1417	1428	1428	1428	1632	1912	1912
n_real_updates/float	1433	1433	1433	1492	1492	1504	1504	1504	1701	1949	1949
petrinet	5761	5761	5761	5761	5761	5761	5761	5761	5761	5761	5695
prime	20196	12218	12218	12218	20930	20250	40220	34549	19995	29482	29482

Table A.2: (continued)

Name	100%	90%	80%	70%	60%	50%	40%	30%	20%	10%	0%
qurt	4545	4545	4692	4692	4692	4722	4727	4727	4779	7751	7669
searchmultiarray	15940	17025	17025	16660	16633	32903	32903	32903	32903	32903	32903
select	5981	5900	11169	11169	11169	11169	11169	11169	11169	11169	11169
selection_sort	2510357	2778554	5468653	5468653	5468653	5468653	5468653	5468653	5468653	5468653	5022053
sqrt	8564	8564	8564	8564	8542	8542	13752	13752	13752	13752	13730
st	356885	287769	351976	355174	420996	432397	436521	395864	508023	463868	504767

APPENDIX B

BENCHMARK PROPERTIES

Shortcut	Description
LOC	Lines of code
NumCond	Number of conditionals
NumLoop	Number of loops
NumFun	Number of functions
NumBB	Number of basic blocks
NumIns	Number of instructions
STot	Total instruction size (byte)
SFmin	Size of smallest function (byte)
SFmax	Size of largest function (byte)
SFavg	Average function size (byte)
SBmin	Size of smallest basic block (byte)
SBmax	Size of largest basic block (byte)
SBavg	Average size of basic blocks (byte)

Table B.1: Basic benchmark properties

Name	LOC	NumCond	NumLoop	NumFun	NumBB	NumIns	STot	SFmin	SFmax	SFavg	SBmin	SBmax	SBavg
adpcm_decoder.c	774	17	13	15	99	817	2522	16	1098	180	2	302	27
adpcm_encoder.c	824	21	15	16	121	872	2694	16	1140	168	2	302	24
adpcm_g721_board_test.c	906	91	11	25	417	1478	5002	24	4750	1000	2	80	12
adpcm_g721_verify.c	918	92	11	25	420	1486	5032	24	4780	1006	2	80	12
binarysearch.c	117	2	1	2	12	39	130	10	120	65	2	26	13
bsort100.c	119	3	3	3	17	57	168	24	102	56	2	30	9
cjpeg_jpeg6b_transupp.c	1216	16	52	1	166	2513	8276	8276	8276	8276	2	380	49
cjpeg_jpeg6b_wrbmp.c	522	41	11	4	55	204	638	38	370	159	2	84	11
complex_multiply.c/float	56	0	0	2	14	86	250	50	200	125	4	54	17
complex_multiply.c/fixed	55	0	0	2	4	59	184	24	160	92	4	112	46
complex_update.c/float	76	0	2	2	21	213	588	140	448	294	2	182	28
complex_update.c/fixed	74	0	0	2	4	93	244	16	228	122	4	154	61
compressdata.c	381	11	4	12	20	121	418	86	226	139	2	88	24
convolution.c/fixed	67	0	2	2	7	48	128	32	96	64	2	62	18
convolution.c/float	69	0	4	2	17	76	208	84	124	104	2	62	12
countnegative.c	135	2	4	6	18	87	260	12	112	43	2	60	14
cover.c	274	0	3	4	399	778	2670	28	1674	667	4	10	6
crc.c	158	6	3	5	27	188	608	68	466	202	4	82	23
dot_product.c/fixed	78	0	1	2	6	40	100	8	92	50	2	42	16
dot_product.c/float	81	0	2	2	13	73	198	70	128	99	2	42	15
duff.c	95	0	2	3	24	99	270	32	196	90	2	44	11
edn.c	326	1	12	9	45	1045	2788	26	1112	309	2	834	61
epic.c	1334	23	47	18	177	2411	7788	90	5142	1298	2	450	44
expint.c	117	3	3	3	21	246	800	14	718	266	2	190	38
fac.c	53	1	1	2	8	25	60	28	32	30	4	14	7
fdct.c	238	0	2	2	8	573	1826	26	1800	913	2	866	260
fft_1024_13.c	449	5	11	7	64	365	1060	28	618	151	2	432	17
fft_1024_7.c	390	5	10	7	62	354	1030	28	588	147	2	430	17
fft_16_13.c	291	5	11	7	64	360	1038	26	606	148	2	428	16
fft_16_7.c	253	5	10	7	62	350	1012	26	580	144	2	426	16
fft1.c	246	7	11	6	119	626	1888	124	1764	944	2	194	16
fibcall.c	90	0	1	2	7	19	52	14	38	26	2	16	7
fir.c/float	79	0	3	2	18	95	252	76	176	126	2	36	14
fir.c/fixed	76	0	2	2	8	66	172	34	138	86	2	42	21
fir.c/MRTC	329	6	2	2	14	111	330	50	280	165	2	74	27
fir2dim.c/fixed	158	0	13	2	28	253	724	190	534	362	2	110	25
fir2dim.c/float	159	0	13	2	72	384	1084	476	608	542	2	106	15
g721_encode.c	1521	58	9	26	209	1043	3204	18	1580	213	2	82	15
g723_encode.c	1535	58	9	26	209	1044	3198	18	1580	213	2	82	15
gsm_decode.c	1940	47	24	62	334	1921	6120	6	1992	612	2	1962	18

Table B.1: (continued)

Name	LOC	NumCond	NumLoop	NumFun	NumBB	NumIns	STot	SFmin	SFmax	SFavg	SBmin	SBmax	SBavg
gsm_encode.c	3178	93	68	70	678	5343	16374	6	3810	1169	2	2200	24
h263.c	179	0	7	5	69	688	2184	686	1498	1092	2	516	31
h264dec_ldecode_block.c	1509	63	107	4	166	3087	10566	196	5112	2641	2	760	65
h264dec_ldecode_macroblock.c	609	43	13	5	41	986	3360	186	3174	1680	2	662	81
hamming_window.c	100	0	3	1	9	64	186	186	186	186	2	76	26
iir_biquad_N_sections.c/float	116	0	5	2	22	115	286	68	218	143	2	34	13
iir_biquad_N_sections.c/fixed	114	0	5	2	10	116	312	48	264	156	4	170	31
iir_biquad_one_section.c/float	84	0	0	2	15	79	210	10	200	105	2	32	14
iir_biquad_one_section.c/fixed	83	0	0	2	5	50	132	4	128	66	4	86	26
insertsort.c	143	4	2	1	5	67	212	212	212	212	4	84	42
janne_complex.c	131	6	2	2	15	31	94	14	80	47	2	16	6
jfdctint.c	441	8	4	6	10	510	1604	76	1528	802	2	612	160
lcdnum.c	118	3	1	2	41	98	328	60	268	164	2	16	8
lms.c/foat	155	0	3	2	24	185	528	88	440	264	2	132	22
lms.c/fixed	155	0	3	2	10	146	422	56	366	211	2	132	42
lms.c/mrtc	290	8	10	8	145	632	1750	304	1446	875	2	48	12
ludcmp.c	204	5	11	3	42	498	1586	276	1310	793	2	160	37
matmult.c	152	0	5	6	20	134	396	12	216	66	2	144	19
matrix1.c/float	132	0	6	2	22	107	302	90	212	151	2	46	13
matrix1.c/fixed	131	0	6	2	16	92	260	70	190	130	2	46	16
matrix1x3.c/float	96	0	4	2	17	77	200	78	122	100	2	34	11
matrix1x3.c/fixed	70	0	2	1	5	38	94	94	94	94	4	30	18
matrix2.c/float	131	0	6	2	17	123	348	70	278	174	2	54	20
matrix2.c/fixed	132	0	6	2	25	145	408	90	318	204	2	56	16
md5.c	600	5	13	26	68	2099	6354	10	5716	706	2	5286	93
minver.c	223	9	17	4	75	400	1242	38	808	310	2	92	16
mpeg2.c	69204	101	5	14	262	3267	100448	10190	90258	50224	2	90116	384
n_complex_updates.c/float	90	0	2	2	25	250	662	178	484	331	2	100	26
n_complex_updates.c/fixed	89	0	2	2	9	194	518	126	392	259	2	166	57
n_real_updates.c/float	72	0	2	2	9	121	346	70	276	173	2	118	38
n_real_updates.c/fixed	73	0	2	2	15	139	398	106	292	199	2	118	26
ndes.c	497	28	12	5	73	694	2170	72	884	434	2	120	30
petrinet.c	901	55	1	1	182	1492	5358	5358	5358	5358	4	98	29
prime.c	69	2	1	5	19	77	222	14	80	44	6	36	11
qsort-exam.c	155	5	6	3	32	227	690	12	678	345	4	58	22
qurt.c	186	8	1	4	56	263	772	38	456	193	2	42	13
real_update.c/fixed	61	0	0	2	4	31	76	8	68	38	4	38	19
real_update.c/float	66	0	0	2	10	66	172	52	120	86	4	48	17
recursion.c	120	6	0	2	9	24	58	10	48	29	4	12	6
rijndael_decoder.c	159	7	4	2	43	217	712	354	358	356	2	106	16
rijndael_encoder.c	223	10	6	4	54	285	946	256	350	315	2	124	17
searchmultiarray.c	605	17	4	2	15	59	194	10	184	97	2	86	14
select.c	136	11	4	3	41	299	938	938	938	938	2	164	22
selection_sort.c	100	2	2	3	17	53	150	12	138	75	2	46	9
sqrt.c	117	5	2	3	29	112	312	38	178	104	2	42	10
st.c	184	6	5	10	81	354	960	12	246	96	2	60	11
startup.c	199	2	6	2	28	107	290	2	288	145	2	22	10
statemate.c	1223	82	1	12	423	2495	8500	16	2958	1062	2	728	20
test3.c	4126	0	121	122	585	6159	18068	24	152	148	2	114	30

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

BIBLIOGRAPHY

[Abs06] AbsInt Angewandte Informatik, Saarbrücken, Germany. *Worst-Case Execution Time Analyzer aiT for TriCore*, December 2006.

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 243–253, New York, NY, USA, 2001. ACM.

[AMF+04] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM.

[AP03] Alexis Arnaud and Isabelle Puaut. Towards a Predictable and High Performance Use of Instruction Caches in Hard Real-Time Systems, 2003.

[App97] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, 1997.

[BR06] Claire Burguière and Christine Rochange. History-based Schemes and Implicit Path Enumeration. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. McGraw-Hill Science/Engineering/Math, July 2001.

[Cor08] Daniel Cordes. Schleifenanalyse für einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib. Master's thesis, Dortmund University of Technology, Department of Computer Science 12, 2008.

[CP01] Antoine Colin and Isabelle Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *In Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, 2001.

[CPIM05] Antonio Marti Campoy, Isabelle Puaut, Angel Perles Ivars, and Jose Vicente Busquets Mataix. Cache Contents Selection for Statically-Locked Instruction Caches: An Algorithm Comparison. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society.

[DP07] Jean-Francois Deverge and Isabelle Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 179–190, Washington, DC, USA, 2007. IEEE Computer Society.

[Duf83] Tom Duff. Duff's device, 1983. `http://www.lysator.liu.se/c/duffs-device.html`.

[EES+99] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry-strength worst case execution time analysis, 1999.

[EES00] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. Comparing Different Worst-Case Execution Time Analysis Methods, 2000.

[EES02] Andreas Ermedahl, Jakob Engblom, and Friedhelm Stappert. A Unified Flow Information Language for WCET Analysis, 2002.

[EKJ+06] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM.

[Eng97] J. Engblom. Worst-case execution time analysis for optimized code, 1997.

[Fer04] Christian Ferdinand. Worst Case Execution Time Prediction by Static Program Analysis. In *IPDPS*, 2004.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27:68–76, 1992.

[FL06] Heiko Falk and P. Lokuciejewski. Design of a WCET-Aware C Compiler, 2006.

[Fou08] Free Software Foundation. The GNU Project, 2008. `http://www.gnu.org/`.

[FPT07] Heiko Falk, Sascha Plazar, and Henrik Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 143–148, New York, NY, USA, 2007. ACM.

[GEE07] Jan Gustafsson, Andreas Ermedahl, and Jakob Engblom. SWEET (SWEdish Execution Time tool) . Mälardalen University, 2007. http://www.mrtc.mdh.se/projects/wcet/sweet.html.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.

[HP02] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, May 2002.

[ICD05] ICD – Informatik Centrum Dortmund, Dortmund, Germany. *ICD-C Compiler framework Developer Manual* , May 2005.

[Inf07] Infineon Technologies AG. *Tricore - 32-bit Unified Processor Core Embedded Applications Binary Interface (EABI)* , February 2007. `http://www.infineon.com`.

[JW99] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36, 1999.

[Kir03] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.

[KK08] Uday P. Khedker and Bageshri Karkare. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In *CC*, pages 213–228, 2008.

[Knu73a] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1973.

[Knu73b] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[KP04] Raimund Kirner and Peter Puschner. calc_wcet_167 - A WCET analysis framework. TU Vienna, 2004. `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`.

[KS86] Eugene Kligerman and Alexander D. Stoyenko. Real-time Euclid: a language for reliable real-time systems. *IEEE Trans. Softw. Eng.*, 12(9):941–949, 1986.

[KW98] Timothy Kong and Kent D. Wilken. Precise register allocation for irregular architectures. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 297–307, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.*, 30(11):88–98, 1995.

[Lok05] Paul Lokuciejewski. Design and Realization of Concepts for WCET Compiler Optimization. Master's thesis, Dortmund University of Technology, Department of Computer Science 12, 2005.

[Mar03] Peter Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[PK89] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[PP07] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[Pro07] GNU Project. *Documentation for binutils*, 2.18 edition, August 2007. `http://sourceware.org/binutils/docs/ld/index.html`.

[Pua06] Isabelle Puaut. WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[RND+05] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, and Richard B. Brown. Compiler Managed Dynamic Instruction Placement in a Low-Power Code Cache. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 179–190, Washington, DC, USA, 2005. IEEE Computer Society.

[Ros77] Barry K. Rosen. High-level data flow analysis. *Commun. ACM*, 20(10):712–724, 1977.

[Rot08] Felix Rotthowe. Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung . Master's thesis, Dortmund University of Technology, Department of Computer Science 12, 2008.

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, June 1998.

[Sch07] Daniel Schulte. Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler . Master's thesis, Dortmund University of Technology, Department of Computer Science 12, 2007.

[SGW+02] Stefan Steinke, Nils Grunwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, and Peter Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, New York, NY, USA, 2002. ACM.

[SMRC05] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[Ste02] Stefan Steinke. *Untersuchung des Energieeinsparpotenials in eingebetteten Systemen durch energieoptimierende Compilertechnik*. PhD thesis, Universität Dortmund, 2002.

[SWLM02] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[Tid08] Tidorum Ltd. *Bound-T timing analysis tool - Userguide*, February 2008. `http://www.tidorum.fi/bound-t/`.

[TIS95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification* , May 1995. http://refspecs.freestandards.org/elf/.

[Ull73] Jeffrey D. Ullman. Fast Algorithms for the Elimination of Common Subexpressions. *Acta Inf.*, 2:191–213, 1973.

[Ver06] Manish Verma. *Advanced Memory Optimization Techniques for Low-power Embedded Processors*. PhD thesis, Dortmund University of Technology, Department of Computer Science 12, 2006.

[VWM04a] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-Aware Scratchpad Allocation Algorithm. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21264, Washington, DC, USA, 2004. IEEE Computer Society.

[VWM04b] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, New York, NY, USA, 2004. ACM.

[WEE+07] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Muller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-209/2007-1-SE, Mälardalen University, March 2007.

[Weg99] Ingo Wegener. *Theoretische Informatik - eine algorithmenorientierte Einführung (2. Auflage)*. Teubner, 1999.

[Wei93] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, 1993.

[Wil05] Reinhard Wilhelm. Determining Bounds on Execution Times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.