Advanced Compiling Techniques to reduce RAM Usage of Static Operating Systems

Der Technischen Fakultät der
Universität Erlangen-Nürnberg
zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Volker Barthelmann

Erlangen – 2004

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	07.01.2004
Tag der Promotion:	11.03.2004
Dekan:	Prof. Dr. Albrecht Winnacker
Berichterstatter:	Prof. Dr. Michael Philippsen
	Prof. Dr. Wolfgang Schröder-Preikschat

Zusammenfassung

In den letzten Jahren wurde eine steigende Anzahl kleiner eingebetteter Systeme in hohen Stückzahlen eingesetzt. Beispielsweise in der Automobilindustrie, wo annähernd 100 elektronische Steuergeräte (ECUs) in einzelnen Oberklassefahrzeugen und bereits mehrere Dutzend in Mittelklassefahrzeugen verbaut werden. Meist werden kleine "System-on-Chip" Mikrocontroller mit statischen Betriebssystemen benutzt. Da das RAM auf diesen Chips sehr teuer ist und nur wenige KB davon auf solchen Systemen verfügbar sind, ist die Reduzierung des RAM-Verbrauchs ein wichtiger Punkt um Kosten zu senken — besonders bei der Produktion hoher Stückzahlen.

Diese Dissertation stellt einige neue Verfahren vor, um den RAM-Verbrauch solcher Systeme durch die Anwendung fortgeschrittener Übersetzungs- und Optimierungstechniken zu reduzieren. Klassische Optimierungen werden hinsichtlich ihrer Auswirkungen auf den RAM-Verbrauch untersucht. Durch geschickte Auswahl von Optimierungsalgorithmen kann der RAM-Verbrauch in einer Testreihe um fast 20% gesenkt werden. Obergrenzen für Stackgrößen der Tasks der Anwendung werden vom Übersetzer statisch berechnet. Durch modulübergreifende Analyse auf Hochsprachenebene werden hier gute Ergebnisse erreicht, die im Vergleich mit einem kommerziell verfügbaren Werkzeug Vorteile in der Handhabbarkeit und Zuverlässigkeit zeigen. Als wichtigster Punkt werden die Registersätze, die das Betriebssystem sichern muss, wenn ein Task unterbrochen wird, optimiert, indem vermieden wird, Register unnötig zu speichern. Registervergabe über Taskgrenzen hinweg reduziert den Speicherbedarf für diese Registersätze weiter.

Die neuen Algorithmen wurden in einen Übersetzer eingebaut und eine kommerzielle OSEK Implementierung wurde modifiziert, um die neuen Optimierungen zu nutzen. Tests auf echter Hardware, sowie Vergleiche mit kommerziellen Programmen zeigen nicht nur, dass das System funktioniert und sowohl Benutzbarkeit als auch Wartbarkeit verbessert, sondern auch, dass eine signifikante Reduzierung des RAM-Verbrauchs und der damit verbundenen Kosten möglich ist. In einer Reihe von Benchmarks wird der RAM-Verbrauch i.d.R. um 30%–60% gesenkt.

Abstract

In recent years, a rapidly growing number of small embedded systems have been used in very high volumes. One example is the automotive industry, where the number of Electronic Control Units (ECU) in a single car is approaching 100 for high end automobiles and several dozens are used in mid-range cars. Small system-on-chip microcontrollers are often used with static operating systems. As on-chip RAM is rather expensive and only few KBs of RAM are available on such devices, reducing the RAM usage is an important objective in order to save costs — especially in high-volume production.

This thesis presents several new approaches to reduce the RAM usage of such systems by applying advanced compilation and optimization techniques. Common optimizations are examined regarding their impact on RAM usage. By selecting classical optimization algorithms regarding their impact on RAM usage, the RAM required for a series of test cases is reduced by almost 20%. Upper bounds for stack sizes of application tasks will be statically calculated using high-level analysis available in the compiler. Comparisons with a commercial tool working on machine-code-level show clear advantages regarding maintainability as well as reliability. Most important, the register sets stored by the operating system when a task is preempted are optimized by abstaining from saving unnecessary registers. Inter-task register-allocation further reduces the RAM required to save those task contexts.

The new algorithms have been added to a production quality compiler and a full commercial OSEK implementation was modified to make use of the new optimizations. Tests on real hardware as well as comparisons with commercial tools not only show that the system works and improves usability and maintainability, but also that significant reductions of RAM requirements, and therefore cost savings, are possible. In a series of benchmarks, RAM usage is reduced on average by 30%–60%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, a rapidly growing number of small embedded systems have been used in very high volumes. One example is the automotive industry where the number of Electronic Control Units (ECU) in a single car is approaching 100 for high end automobiles and several dozen in mid-range cars (an overview of the ECUs in the current 7-series BMW is given in [68]; for numbers of microcontrollers in different cars, see [15]). Another domain is the emerging area of wearable computing which may produce very small systems in high volume.

Typically system-on-chip microcontrollers are used which contain the CPU, peripherals, ROM and RAM on a single chip. ROM sizes of a few hundred KB and RAM sizes of a few KB are common. Figure 1.1 shows a die overlay of the MPC565, a microcontroller currently used in automotive applications. While the flash ROM with 1MB capacity clearly is the largest part of the chip, the blocks of RAM (which are only 4KB each) are the second biggest factor influencing die size, much bigger than, for example, the PowerPC core. When taking capacity into account, it can be observed that RAM takes up about ten times as much chip space compared to programmable flash ROM. This shows that reduction of RAM usage is an important goal. Even transformations that trade off RAM for ROM can still save die size unless about ten times as much additional ROM has to be traded for the RAM that is saved. Furthermore, off-the-shelf chips have a fixed amount of on-chip RAM and ROM. To fit an application into the chip, both ROM as well as RAM has to be sufficient (otherwise a larger and more expensive off-the-shelf chip has to be used). If there is not quite enough RAM space, but some unused ROM space, RAM reduction can help to fit the application on the smaller chip and save costs.

Not very long ago, these small systems were typically programmed in assembly language using no operating systems. In recent years, however, high level languages are often being used for such systems (see, for example, [93] or [135]) and the use of operating systems is getting more common. This is also illustrated by the fact that the market segment of 8bit microcontrollers in 2002 has shrunk by about 10%, whereas the segment of 32bit controllers (that are much better suited to e.g. C compilers) increased about 40% (16bit controllers stayed about equal), see [136], [137]. Also, new 8bit architectures are tailored to C programming, e.g. the Atmel AVR (see [9]). The current roadmap of Motorola [105], market leader for 8bit controllers, only shows future products for their HC08 series that has been optimized for C programs [136].

As, for example, the ECUs in cars are usually connected via several bus systems and have to co-operate, a common communication layer provided by the operating system is a big benefit. In addition, using the standard features of well-tested operating systems seems to promise more stable systems and shorten development times as application

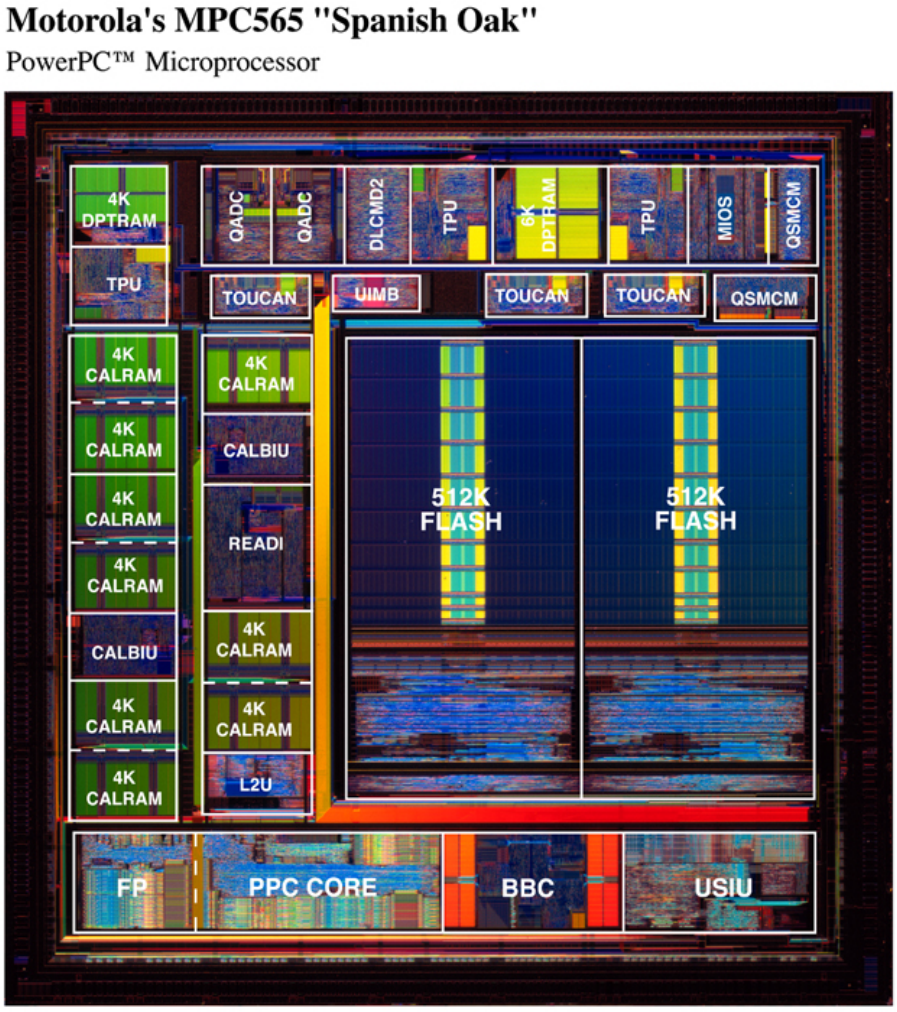


Figure 1.1: Die Overlay showing ROM and RAM sizes

programmers do not have to "re-invent the wheel".

Use of high level languages and operating systems also significantly reduces the amount of work necessary to port an application from one chip or architecture to another. Therefore, chips can be chosen more freely — an important benefit in volume production where every penny counts.

1.1 Static Operating Systems

Classic operating systems are not well suited for such systems because of several reasons. The most important of these being the high memory requirements. Another problem is the danger of fragmentation or memory leaks. Many ECUs are safety-critical and the risk of failure to allocate a system resource due to lack of RAM or memory fragmentation can not be tolerated.

Static operating systems help to avoid these problems. In such operating systems all system resources like tasks, messages, mutexes etc. used by the application are configured statically using an offline tool. After verifying validity, a kernel which is specially optimized to this very application is generated from this configuration (see figure 1.2).

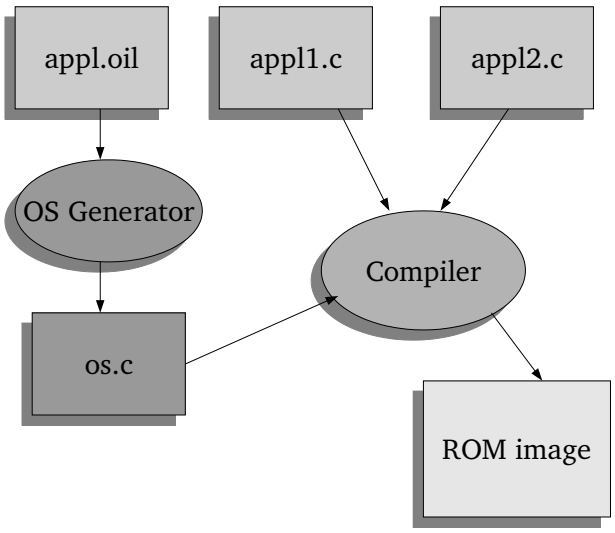


Figure 1.2: Architecture of a Static Operating System

All resources needed by the operating system are statically allocated during generation time and therefore no dynamic memory management is needed. Allocating an operating system resource can never fail!

As all resources used by the application are known when the kernel is generated, it is possible to produce a very small optimized kernel. Functionality not needed by this application can be automatically omitted, and the size of data structures can be reduced. For example, a task can be identified by a single byte that is used as an index into arrays containing the necessary information for all tasks. This is often more efficient than using a (larger) pointer to a task control block as is used in many systems.

Also, it allows the constant parts of the task control block to be placed into ROM. For very small systems, this can be important.

These benefits of static operating systems allow them to be used in very small systems (e.g. 8bit microcontrollers with 32KB ROM and 1KB RAM) and still leave enough room for the application.

The market of static operating systems is currently dominated by systems conforming to the OSEK specification [113] which will be described below.

1.2 Compilers for Embedded Systems

The way software for such systems is being developed has changed over the years. Traditionally, small systems that demand efficient and small code have been programmed almost exclusively in assembly language. The lack of compilers able to produce code that is sufficiently efficient was the main reason. Also, there are special requirements of such code (e.g. interrupt handlers, accessing special hardware etc.) that could not be expressed in high-level programming-languages.

Over time, compiler technology has improved. Many optimizations were built into compilers and — also an important point — the platforms used for developing (i.e. personal computers and workstations) have become powerful enough to run optimizing compilers with reasonable turn-around times.

Contrary to embedded systems, software development for larger systems has been done almost exclusively in high-level languages for a long time. With optimizing compilers becoming more common, assembly-language is being used less and less. Basically, compilers generate code that is "good enough" and there is no return of investment for writing code in assembly language.

However, the constraints for high-volume embedded systems are much tighter. Increased space requirements imply bigger hardware and higher costs. Therefore, there are much more cases where better optimizations allow significant cost reductions — which is an important reason why high-level languages were rarely used.

There are, however, good reasons for using high-level language compilers now. Software developers able to write good assembly code are rare — especially as every architecture has its own assembly language. Also, writing optimized assembly language code is usually more time-consuming and error-prone. As the complexity of embedded systems increases, assembly language becomes harder and harder to maintain. Also, as the market for embedded devices grows (see e.g. [87]), there is a growing demand for more and larger embedded software which has to be satisfied. The market size of embedded systems is about 100 times the desktop market and compilers for embedded systems will become more important according to [51].

Furthermore, code written in assembly language is tied to a certain architecture and usually can not be moved to another one without completely rewriting it. It is often necessary however to reduce costs by moving the application to another chip that is cheaper. With high volumes, the cost savings can be much higher than the cost of a recompilation. For assembly code, the cost of moving the application is much higher. Even if it is still lower than the savings that could be obtained by moving to another architecture, the time required to rewrite the code for the new system will rarely be acceptable.

As a result, there is a need for highly optimizing compilers that are able to generate very efficient code for such systems. Currently available compilers meet these constraints to varying degrees. Classical global optimizations are reasonably common

although there are a few (especially small and somewhat unusual) architectures that lack highly optimizing compilers.

There are countless research projects dealing with various aspects of compilation for embedded systems. The LANCE project [93] concentrates on retargetable code generation for DSPs, the SPAM project [130] examines very expensive optimizations to reduce hardware costs of embedded systems. Research on resource-aware compilation is described in [120]. The OPTIMIST project [111] examines integrated code generation for irregular architectures. Machine-independent optimizations and hardware-software co-design are researched in the OOPS project [112]. The SUIF [138] and Zephyr [153] projects offer frameworks for retargetable optimizing compilers. There are also many other academical and commercial compiling systems for embedded systems.

1.3 RAM Usage

The amount of RAM available on a small system-on-chip is very expensive and typically at least an order of magnitude smaller than on-chip ROM size (see section 2.2). Therefore, minimization of RAM usage is a very important objective — especially as optimizing for RAM usage seems somewhat neglected in existing compiler design.

Usage of RAM in a static system can usually be attributed to the following sources:

- static application data
- application stack
- static operating system data
- task contexts

Static application data can rarely be minimized unless the developer wrote code that contains unused data. Similarly, static operating system data is already minimized at generation time — one of the benefits of using static operating systems.

That leaves the application stacks and task contexts as major stack consumers. Task contexts contain the register sets the operating system must save when a task is preempted. Application stacks generally consist of the following items:

- local variables used by the task code
- register values pushed by the compiler
- temporary variables created by the compiler
- safety margins (if the stack size needed is not known exactly)

1.4 Contributions and Organization of this Thesis

This thesis presents several approaches to reduce RAM usage by applying new advanced compilation and optimization techniques. Chapter 2 introduces the environment that was used to implement and test these optimizations. The OSEK operating system is used as a static operating system. Some typical microcontrollers that are used in the automotive industry are presented, and an overview about vbcc, the compiler that was used as a basis to implement the new optimization algorithms, is given.

Following that, a series of classical compiler optimizations is discussed regarding the applicability to small embedded systems in chapter 3. Special attention is put on the

impact of those optimizations on RAM usage. Test results suggest that optimizing for RAM requires different compiler settings than optimizing for speed or code size. Therefore, an "optimize for RAM" option seems to be a promising extension of compilers for small embedded systems.

Chapter 4 on stack analysis describes a new approach to statically calculate the size of task stacks within the compiler to eliminate safety margins. Stack analysis was built into the compiler and comparing the results with a commercial post link-time tool shows the advantages of the high-level analysis performed.

Reducing the size of task contexts is the objective of chapter 5. The new idea is to compute optimized contexts for every task which contain only the registers that really have to be saved. Furthermore, the register allocation of the compiler is adapted to perform inter-task register-allocation. Experimental results show the theoretical improvements possible with this new optimization.

All these measures are then combined in a real world implementation in chapter 6. A commercial OSEK system is modified to work with the improved vbcc compiler. Context optimization and inter-task register-allocation is performed as well as stack analysis and some additional improvements. Generation of task stacks and task contexts is moved from the operating system generator to the compiler (see figure 1.3). Several problematic attributes like the stack size of tasks no longer have to be specified.

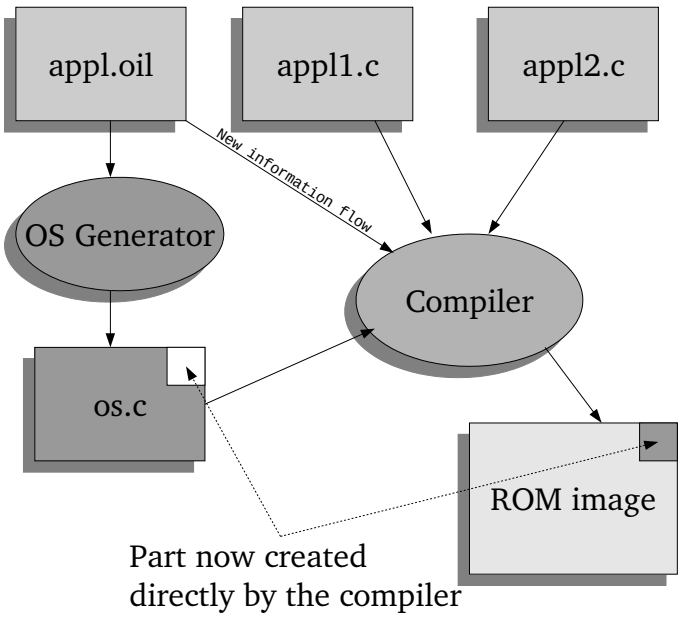


Figure 1.3: New Architecture

Tests on real hardware show not only that the system works and that the usability and maintainability is improved, but also that significant reductions in the RAM requirements are possible.

Chapter 2

Target/Environment

This chapter gives an overview of the target environment that is of interest in this thesis. A short introduction to OSEK OS, perhaps the most important static operating system, is given. Also, a few important microcontrollers that are used with OSEK in high-volume products are presented. Furthermore, the compiler that was used to implement the new optimizations that are proposed in this thesis is described shortly.

2.1 OSEK

The OSEK/VDX project defines an industry standard for an open software architecture for ECUs in vehicles. It is a result of harmonization between the German OSEK ("Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" — Open systems and the corresponding interfaces for automotive electronics) and the French VDX ("Vehicle Distributed eXecutive"). Usually, just the term OSEK is used. OSEK is an open standard which is currently in the process of ISO standardisation (ISO 17356). OSEK is not a certain implementation but it specifies the behaviour and services of operating systems. Several vendors offer OSEK compliant operating systems, and ECUs running such OSEK implementations are used in production cars today. Currently, OSEK compliant operating systems from several companies have been officially certified [1, 144, 102, 95, 151, 57, 3]. Also, there are other commercial and academic implementations (e.g. [157, 133]).

OSEK/VDX consists of several specifications, including Operating System [114], Communication [118], Network Management [117], Time-Triggered Operating System [115], and others. This section will give a short overview of those parts of the OSEK Operating System specification which are relevant for this thesis. For full details, consult the OSEK specifications and the OSEK/VDX homepage [113].

2.1.1 Philosophy

The OSEK OS specification describes an API every OSEK compliant operating system has to implement. Only the behaviour is prescribed, the exact implementation, however, is up to the vendor. The main principles are:

- fully static system
- highly scalable
- portability of application code (source level only)
- configurable error checking

- support for running code from ROM
- only minimum hardware requirements - must run on 8bit controllers
- predictable timing

The static configuration of the operating system is provided in a separate file, see section 2.1.3.

2.1.2 Portability

While portability on source level is desired, compromises have to be made here. Accessing hardware like on-chip peripherals is still done by the application. As very small systems are used on widely different hardware, a general hardware abstraction layer has been considered too expensive. It may be provided as an extension, but it is not required from the operating system.

2.1.3 OSEK Implementation Language

A standard file format called OIL (OSEK Implementation Language, see [116]) is used to specify the tasks, scheduling properties, and all other operating system resources used by the application. Note that this is the language to configure the static operating system parameters, not the language used to write the application code (this is usually done in C). Figure 2.1 shows part of an example OIL file that contains some global properties, a task description, as well as some OSEK events and resources (see below). Apart from standard attributes, an OIL file may contain additional vendor specific attributes that follow the syntax described in the OIL specification.

2.1.4 Task Management

OSEK tasks are the main threads of execution in a system. Although it is not explicitly stated, they share the same address space and there is no memory protection. OSEK does, however, discern between tasks that may enter a waiting state (called "extended tasks") and those that never do ("basic tasks"). This property has to be statically configured for each task to allow for a series of optimizations during generation of the operating system.

The control structures of each task can be statically allocated during system generation as there can only be one instance of every task. Basically, tasks can not be created, only activated. The possible task-states for an extended task are shown in table 2.1:

Table 2.1: OSEK Task States

task-state	**description**
running	The CPU currently executes this task.
ready	The task is ready to enter the *running* state, but another task is currently executing.
waiting	A task is blocked because it requires an event (see below).
suspended	The task is passive and has to be activated.

Figure 2.2 shows the possible task-state transitions of an extended task. For basic tasks, the `waiting` state and its edges do not exist.

```
#include<PPC.oil>
CPU OSEK_PPC
{
  OS ExampleOS
  {
    MICROCONTROLLER = MPC555;
    CC = AUTO;
    SCHEDULE = AUTO;
    STATUS = EXTENDED;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = FALSE;
  };
  TASK ExampleTask
  {
    TYPE = AUTO;
    AUTOSTART = FALSE;
    SCHEDULE = FULL;
    RESOURCE = Opener;
    EVENT = WindowDown;
    EVENT = WindowUp;
  };
  EVENT WindowUp ;
  EVENT WindowDown ;
  RESOURCE Opener
  {
    RESOURCEPROPERTY = STANDARD;
  };
};
```

Figure 2.1: OIL Example

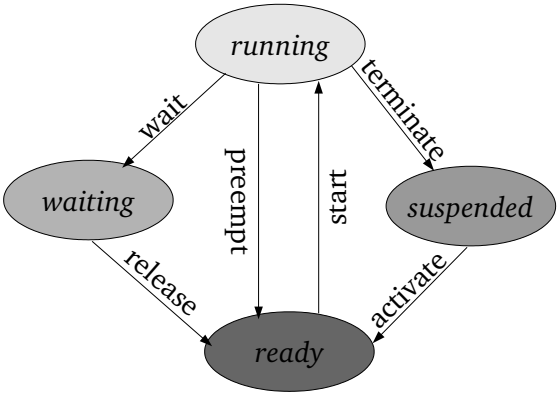


Figure 2.2: OSEK OS Task-State Transitions

Every task has a statically assigned priority as well as an attribute specifying whether the task is fully preemptive or non-preemptive. A fully preemptive task is interrupted immediately when another task with higher priority gets ready (see figure 2.3 which assumes Task 2 has a higher priority). A non-preemptive task may only be interrupted when it voluntarily releases the CPU by invoking a system function like `Schedule()` (therefore, in figure 2.4, Task 1 keeps running even after Task 2 got ready).

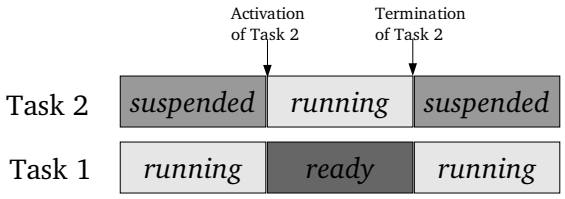


Figure 2.3: OSEK OS preemptive Scheduling

In application code, each task is identified with the `TASK` keyword (a C macro defined in an operating system header file):

```
TASK(myTask)
{
  ...
  TerminateTask();
}
```

Therefore, the code belonging to a task can be easily identified from the source code. While the tasks can have shared code (e.g. when calling the same function from the task body), each task has its own top-level function. Also, there can only be one instance of every task. Activation and termination of tasks is controlled by the following services:

- `ActivateTask()` moves a task from the *suspended* to the *ready* state.

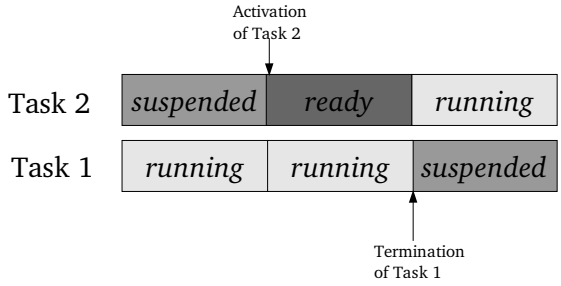


Figure 2.4: OSEK OS non-preemptive Scheduling

- `TerminateTask()` is used to terminate a task. A task can only terminate itself and must not terminate other tasks. A task must not return from its top-level function without calling `TerminateTask()`.

2.1.5 OSEK Events

Events are the OSEK equivalent of semaphores. They are objects statically defined in the OIL file. A task can wait on those events (waiting for several events at once is possible) it has been assigned in the OIL file. Tasks which are not assigned an event ("basic tasks") can never enter the *waiting* state. Setting of events can be done by every task. Every task has its own set of events, i.e. when a task sets an event, it has to specify for which task it is set (this allows sharing of bits in a mask of events). The most important system services related to events are:

- `WaitEvent()` puts a task in *waiting* state until one of the events it is waiting for is set (it will return immediately if one event is already set).
- `SetEvent()` sets an event for a task.
- `ClearEvent()` clears some of the events of the task calling this service.

2.1.6 OSEK Resources

OSEK Resources are mutexes used to implement controlled access to shared resources. To avoid the problems of deadlocks and especially priority inversion (for further information, see e.g. [24, 81]), OSEK implements a "priority ceiling" or "highest locker" protocol. All resource objects are statically assigned to tasks, i.e. any task that may want to obtain a resource must specify this fact in the OIL file. As soon as a task obtains a resource, its priority is raised to the highest priority of all tasks that may require that resource. The relevant system services of OSEK are:

- `GetResource()` obtains the specified resource and possibly increases the task's priority.
- `ReleaseResource()` releases the resource and resets the priority.

While occupying a resource, a task is not allowed to terminate itself or to wait for an event. As a result, a task will never be put into the `waiting` state when trying to obtain a resource — obtaining a resource will always succeed.

2.1.7 Interrupt Service Routines

Embedded systems usually do a lot of work that depends on external inputs, e.g. from sensors or busses. Interrupt service routines are used to handle such external events without the need for polling. As OSEK has to run on very small systems that may have to deal with high interrupt loads, efficiency is important here. Therefore, interrupt handlers in OSEK are very close to the hardware. There are even two general categories of ISRs (interrupt service routines) defined in OSEK:

- ISRs of category 1 are not allowed to call (almost all) system services.
- ISRs of category 2 may call several system services including `ActivateTask()` and `SetEvent()` which can cause rescheduling.

While ISRs of category 1 in general do not affect the operating system, category 2 ISRs usually have to be called in some special environment to allow rescheduling at the right point in time, i.e. just before returning from ISR level to task level. Figure 2.5 shows the right point in time to dispatch a task that has been activated by an ISR (assuming `Task2` has a higher priority and `Task1` is preemptive).

In the application code, those ISRs are written in a similar fashion to tasks:

```
ISR(myISR)
{
  ...
}
```

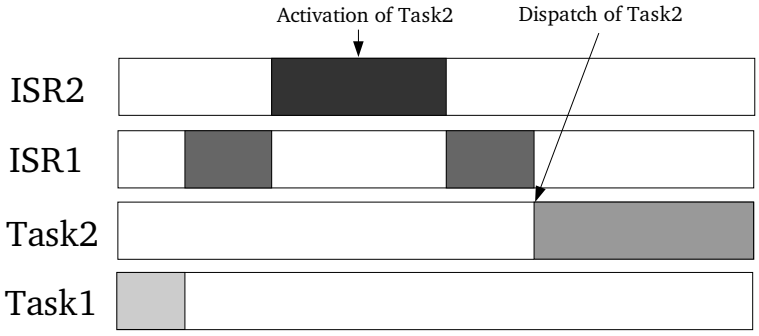


Figure 2.5: OSEK Interrupt Handling

2.1.8 Properties of Application Code

OSEK applications have certain properties that are very important for the work in this thesis, notably:

- no heap memory management
- no recursions
- use of libraries as source code rather than object code
- restricted use of function pointers

These restrictions are strongly suggested, for example, by the OSEK OS specification [113] or guidelines published by the Motor Industry Software Reliability Association [103]. As most automotive applications have real-time requirements and are often safety-critical, such constructs should be avoided. For example, recursions and heap memory management make it too difficult to verify that a system does not run out of space at some time. Use of function pointer variables is also considered dangerous. Availability of source code is required for scalability, certification, and portability.

For the compilation and analysis techniques presented in this thesis, these restrictions are crucial. They allow for much better static analysis than dynamic systems that make heavy use of, for example, recursions and heap memory management.

2.2 Examples of relevant Microcontrollers

The following sections will give a short overview of three microcontroller families that have been used for some experiments in this thesis (because of their widespread use in the automotive industry and support of the tools examined in this thesis, i.e. vbcc and StackAnalyzer). All of them are available in versions suitable for the automotive industry, i.e. as systems-on-chip with CPU, peripherals, RAM, and flash ROM on a single die. They are all supported by more than one commercial OSEK implementation and all of them are built into cars out on the road now.

2.2.1 68HC12/HCS12

The 68HC12 produced by Motorola is a small to medium 16bit architecture that has its roots in the old 6800 series of 8bit processors. Many derivates are available, offering different sets of peripherals and varying on-chip memory sizes. There is a line of derivates specially designed for automotive applications. A new, compatible product line, the HCS12 (or Star12) microcontrollers, show the continuing demand. A big field of application is body electronics in current cars (e.g. window openers, dashboard control, etc.).

The on-chip memory available on current 68HC12 devices ranges from 1KB RAM and 32KB of flash ROM up to 8KB of RAM and 128KB of ROM. The new HCS12 series ranges from 2KB RAM/64KB ROM up to 12KB RAM/256KB ROM. Unaligned memory access is allowed without any restrictions, therefore, eliminating the need for padding.

Register Set

The 68HC12/HCS12 microcontrollers have a small register set that also shows the heritage of their 8bit ancestors. Two 8bit accumulator registers (`A` and `B`) can be used together as one 16bit accumulator (`D`) in many instructions. Additionally, there are two 16bit index registers (`X` and `Y`), a dedicated 16bit stack pointer (`SP`) as well as a program counter (`PC`). An 8bit flag register (`CCR`) stores condition codes and processor

state (see figure 2.6). Segment registers must be used to access the entire ROM area of the devices with larger flash memories (outside of 16bit address space).

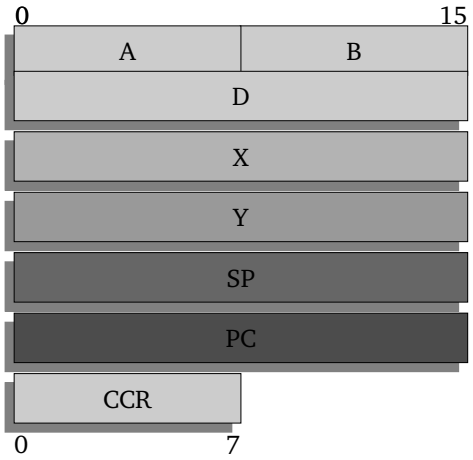


Figure 2.6: HC12 Register Set

Instruction Set

The 68HC12/HCS12 provides instructions of varying length with many different kinds of memory operands available, e.g. absolute addressing, indexed addressing (via `X`, `Y`, or `SP`), double indirection, or pre- and post-increment/decrement addressing modes. As it is an accumulator architecture, most instructions use the accumulator as implicit operand and destination. Therefore, many instructions have a single operand or none at all.

Instructions do not have to be aligned in memory and, in fact, there are instructions occupying every length from a single byte up to six bytes. A so-called "instruction queue" is used to efficiently fetch instructions. No classic pipeline is used, however.

2.2.2 C16X/ST10

The C16X family of 16bit microcontrollers was originally developed by Siemens and various derivates are now produced by several manufacturers, most notably Infineon and ST Microelectronics (under the ST10 label). Many different configurations, varying peripherals as well as memory sizes are available. Typical automotive configurations from Infineon range from 2KB RAM/64KB ROM up to 8KB RAM/256KB ROM. ST Microelectronics offers devices up to 18KB RAM and 512KB ROM. Smaller configurations are used for body control in cars, but higher processing power makes them also suitable for more demanding applications. Classic implementations offer a four-stage pipeline, but improved versions with a five-stage pipeline are also now available.

Register Set

The C16X architecture provides several windows of 16 16bit general purpose registers as well as a series of special purpose registers (see figure 2.7). The general purpose registers are not real core registers but they refer to locations in the internal RAM. A special register, the context-pointer `CP`, contains the start address of the register window in internal RAM that contains the general purpose registers. By changing

the context-pointer, the register window is moved. Basically, register `Rn` is a shortcut for the word at memory location $\texttt{CP} + 2 \cdot \texttt{n}$. For compilers and operating system, the following registers are of interest:

- `SP`: A dedicated stack pointer is available. Return addresses and interrupt frames are stored through this register. It can only address a limited part of internal RAM.
- `STKOV/STKUN`: These registers can be used for hardware monitoring of the internal stack. If the stack pointer leaves the window specified by those registers, an exception is raised.
- `CP`: With this context pointer, the window of general purpose registers can be moved around in the internal memory.
- `PSW`: The processor status word contains condition codes, interrupt levels etc.
- `MDL/MDH`: To reduce interrupt latency times, slow multiplication instructions can actually be preempted by interrupts. These registers store the state of the multiplication unit.

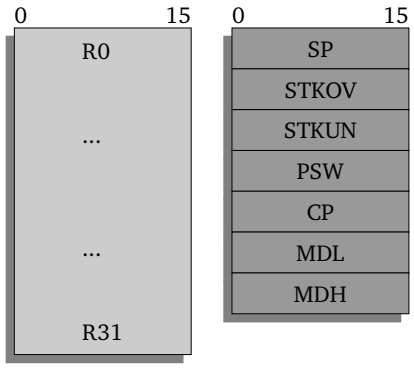


Figure 2.7: C16X/ST10 Register Set

To make use of more than 64KB of address space, segment registers must be used.

Instruction Set

All instructions on the C16X occupy either 16 or 32 bits and have to be aligned on word boundaries. Many instructions have two operands with one operand being both a source and destination. While in most cases one operand must be a general purpose register, the other one can also be a (real) memory operand or an immediate constant. Addressing modes include absolute addressing, indirect addressing via a general purpose register, and, in some cases, pre/post-increment/decrement modes.

As the dedicated stack pointer register is not usable in indexed addressing modes and can only address a part of internal RAM, a general purpose register is commonly used by compilers as a so called “user” stack pointer to address local variables etc.

2.2.3 MPC5XX

The Motorola MPC5XX family is a series of 32bit RISC microcontrollers designed for the automotive industry. They adhere to the PowerPC ISA (instruction set architecture) that has its roots in the IBM POWER architecture of high-end RISC processors for servers and workstations. There are also various other implementations of the PowerPC architecture made by other manufacturers [7, 74]. They range from microcontrollers to high-end server CPUs. The MPC5XX series is at the lower end of this range.

It is, however, on the higher end of microcontrollers currently using static operating systems. A big field of application is sophisticated engine management. Its high processing power (compared to the other microcontrollers described here) as well as some special peripheral support (e.g. programmable timer units) allow the ability to control high-revving engines with a larger amount of cylinders. Also, complex calculations required e.g. for knock detection can be implemented.

The on-chip memory sizes of current devices range from 26KB RAM/448KB ROM up to 36KB RAM/1024KB ROM.

Register Set

The MPC5XX derivates have 32 64bit floating point registers, 32 32bit general purpose registers, an instruction pointer (32bit), and a series of (partially device-specific) special purpose registers (which can be transferred to/from general purpose registers but not directly to memory). This is illustrated in figure 2.8. The special purpose registers that are of interest for compilers and/or operating systems are:

- `CR0-CR7`: A 32bit register that stores 8 sets of 4bit condition codes.
- `LR`: When calling a subroutine, the return address is stored in this link register. Also, it can be used as the target of a computed call or jump.
- `CTR`: The special counter register can be used in some loop instructions or specify the target of a computed call or jump.
- `FPSCR`: Floating point status (e.g. rounding mode) is stored in this register.
- `XER`: Integer exception information is stored in this register.
- `MSR`: The machine state is stored in this register. It contains, for example, the interrupt enabling flag.
- `SRR0/1`: When servicing an exception, the CPU stores the old program counter and machine state register in those registers.
- `SPRG0-3`: These four registers are reserved for the operating system. They are needed, for example, to provide nested interrupt handlers.

Instruction Set and Execution Units

As a RISC architecture, all the PowerPC instructions occupy 32 bits. Most arithmetic instructions have three operands. Only registers and small immediate constants are available as operands. All accesses to memory have to be done via special load and store instructions. Addressing modes supported are indirect (through any general purpose register) with constant 16bit offset, register offset, or pre-increment/decrement modes.

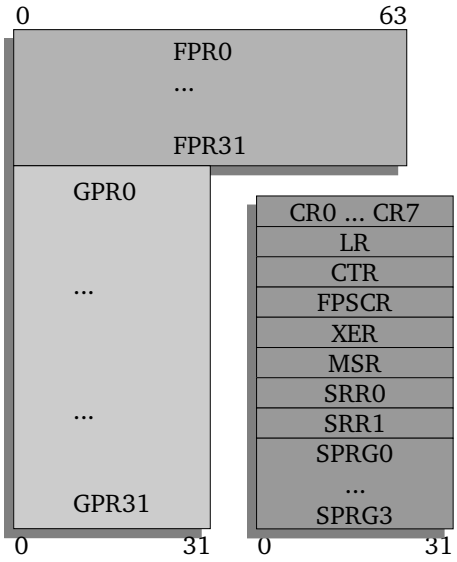


Figure 2.8: PowerPC Register Set

All instructions have to be aligned (as well as memory accesses). An instruction cache is available and a four-stage pipeline allows single-cycle execution of many instructions. The CPU provides four execution units:

- The *integer unit* performs arithmetic operations on the general purpose registers, etc.
- Floating point operations are executed in the *floating point unit.*
- The *load/store unit* handles all accesses to memory.
- Changes in control flow are handled by the *branch unit* which supports, for example, static branch prediction.

The units can operate somewhat in parallel and also some out-of-order execution is possible. Nevertheless, the CPU appears fully sequential to the application. For example, zero-cycle branches are possible.

2.3 The vbcc Compiler

This section shall give a short overview of vbcc, a compiler that was written by the author of this thesis and is used as a basis for the work that is presented herein.

2.3.1 General

vbcc is a highly optimizing portable and retargetable ISO C compiler. It supports ISO C according to ISO/IEC 9899:1989 and a subset of the new standard ISO/IEC 9899:1999 (C99).

It is split into a target-independent and a target-dependent part, and provides complete abstraction of host- and target-arithmetic. Therefore, it fully supports cross-compiling for 8, 16, 32 and 64bit architectures.

The compiler itself is written in portable C and can be compiled and run on all common host operating systems like Linux, Unix, or Windows.

While it is not sold commercially at the moment, the compiler itself is not an academic research project but a production quality compiler that has been in real public use and has successfully translated large projects, including itself, as a search of the web or newsgroups for `vbcc` and `compiler` will reveal (see [54] for a visually appealing application using vbcc).

The compiler translates C files to assembly code (several backends exist). A frontend with Unix `cc` compatible options is provided that calls compiler, assembler, scheduler, and linker. This frontend is driven by a configuration file and enables support for different target systems or assembler/linker tool chains.

2.3.2 Support for Embedded Systems

When programming embedded systems, there can be several additional requirements for compilers. The next paragraphs will give a short description of what features are available in vbcc and how they are implemented.

Target-Specific Extended Types

Obviously, in a retargetable compiler, the size and layout of standard data types must be controlled by the backend and also adhered to by the frontend. For example, the code

```
if(65535U + 1 == 0)
  a();
else
  b();
```

has to call `a()` on a target with 16bit `int` and `b()` on a target with 32bit `int`, no matter what size `int` is on the host.

This is not specific to embedded systems but rather to any retargetable compiler able to work as a cross-compiler. For many embedded architectures, however, further non-standard data-types are required.

As an example, many microcontrollers have special instructions supporting single bits. To make efficient use of them, bit types must be supported. Also, many 16bit controllers have segment registers that allow accessing more than 64KB of memory. However, as updating the segment registers for any memory access is very expensive, often code and/or data may be placed in special "near" sections which avoids updating of the segment registers. As a result, a pointer to such a "near" object may occupy one word and fit in a single register whereas a pointer to a "far" object requires two words and registers. A frequent case on small systems is to have short pointers to data and large pointers to functions.

vbcc allows backends to extend the type system of the frontend. This is used, for example, in the C16X/ST10 backend to provide different pointer sizes and a bit type.

Section Attributes

When programming an embedded system, some objects usually have to be placed in specific memory locations. For example, an interrupt service routine may have to be located at a specific address. vbcc provides an attribute that will put an object into a specified section (if supported by the backend and object format) which can then be located by the linker:

```
__section(".reset") reset_handler()
{
...
}
```

Interrupt Handlers

Interrupt service routines will be needed in almost every embedded system. While interrupt handlers for operating systems like Unix are mostly normal C functions that are called by the operating system, interrupt handlers for embedded systems are often directly called by the hardware.

As they are not called by a C function, there are other rules for these functions than the standard application binary interface (ABI). For example, returning from an interrupt is usually done with a different machine instruction than returning from a subroutine. Also, subroutines may destroy certain register contents as the calling function knows that they may be overwritten. However, an interrupt can preempt the running code at any time. Therefore an interrupt handler must not destroy any registers.

Special compiler support is needed to be able to write interrupt handlers in C rather than assembly language. If supported by the backend, vbcc provides a special attribute to designate an interrupt handler:

```
__interrupt isr_handler()
{
...
}
```

Inline Assembly

While most parts of an embedded system can usually be written in C, there are almost always some exceptions. For example, accessing a peripheral module on the microcontroller may require a special machine instruction the compiler does not emit (because it is not useful for standard C code).

Of course it is always possible to write a function in assembly language adhering to the ABI required by the compiler and call this code from C. However, this has some disadvantages:

- The ABI of the compiler may not be completely documented or hard to understand (e.g. rules for argument passing).
- Often, only a single machine instruction is needed. Creating an extra file to write an assembly function containing the instruction is tedious.
- Calling a separately assembled function will incur some overhead of execution time, code space, and RAM (return address).
- A function unknown to the compiler will be called and may cause significantly worse code generation. For example, the compiler may have to assume that the function destroys register contents or some variables.

To overcome these problems, many compilers provide some facility to include inline assembly code inside the C source. Usually this code is pasted more or less directly

into the output generated by the compiler. What is common amongst most compilers is that they do not parse or understand the inline assembly code and they assume that it behaves similar to a function call (e.g. jumping from one piece of inline assembly to another one and returning there will confuse most compilers). The syntax differs widely among different compilers as well as some of the following interesting points:

- Is there a way to pass arguments to the assembly code?
- Can assembly code use some temporary registers without risk that they might collide with registers used to pass arguments?
- What optimizations can the compiler do to the inline assembly code (e.g. function unrolling)?
- Does the compiler assume worst-case side-effects or is it possible to specify the side-effects caused by the inline assembly?

vbcc provides inline assembly through an extended syntax for function declarations. Internally, inline assembly is handled like a function call in vbcc. This automatically allows special features available for functions to be applied to inline assembly without the need for extra work. For example, the following declaration specifies an inline assembly function `sin` which uses only register `fp0` (`__regsused("fp0")`) and receives its argument in register `fp0`. Furthermore, the assembly code to insert is `" fsin fp0"`.

```
__regsused("fp0") double sin(__reg("fp0") x)=" fsin fp0";
```

2.3.3 Optimizations

vbcc performs a series of classical optimizations. Data-flow analysis is performed. The following list shows the most important optimizations that are performed:

- cross-module function-inlining
- partial inlining of recursive functions
- inter-procedural data-flow analysis
- inter-procedural register-allocation
- register-allocation for global variables
- global common-subexpression-elimination
- global constant-propagation
- global copy-propagation
- dead-code-elimination
- alias-analysis
- loop-unrolling
- induction-variable elimination
- loop-invariant code-motion
- loop-reversal

2.3.4 Cross-Module Analysis

The new optimizations presented in this thesis require analysis of an entire application or at least an entire task. As the source code for tasks in embedded systems often is spread across several modules (source files), a compiler must not restrict its scope to single files to implement these optimizations.

vbcc is able to read in several source files at once and analyze and translate them as if the entire source code is contained in just one file. Its internal data structures allow it, for example, to discern between two static variables of the same name if they come from different source files.

Many projects, however, are built on makefiles that expect separate compilation. For example a makefile for a simple project consisting of two source files `t1.c` and `t2.c` could look like this:

```
t1.o: t1.c
    $(CC) $(CCOPTS) -c -o t1.o t1.c

t2.o: t2.c
    $(CC) $(CCOPTS) -c -o t2.o t2.c

test: t1.o t2.o
    $(LD) $(LDOPTS) -o test t1.o t2.o
```

Such a procedure is also supported by using the frontend of vbcc. If a high optimization level is specified for the frontend of vbcc, it will pass an option to vbcc, telling it not to generate a normal object file. Instead vbcc will create a pseudo object file that retains enough information from the source file to delay analysis and optimization to some point later in time.

In the linking phase, the frontend will detect these pseudo objects and, instead of passing them directly to the linker, will call vbcc once again with all the pseudo objects to be linked at once. vbcc will optimize and translate them and the resulting object will be linked together with all the "normal" objects to produce the final executable. Similar schemes are used by other compilers, e.g. [128].

The makefiles will still work although every call of `make` will basically result in a complete rebuild as all the real work is now done in the linking phase which is always executed. However, with this mechanism switching from normal (and fast) separate compilation to cross-module optimization can be done just by changing the optimization settings for the frontend. There are only minor restrictions, e.g. it is not possible to use different options for each module as they are translated together.

Chapter 3

Common Optimizations

In this chapter, common standard optimizations available in current compilers and frequently presented in textbooks and scientific papers are discussed. After an introduction which discusses the usefulness of optimizing compilers in general, a series of typical optimizations performed by optimizing compilers are presented.

Many of the typical optimizations have been developed for personal computers, workstations and servers. In fact, most of the scientific research in the area of compiler optimizations has targeted such machines — issues like code-size or memory requirements of the generated code have only recently gained attention once more. Therefore, the question of whether the optimizations discussed are also recommendable for small embedded systems is examined (mostly they are) and which of them are suitable to reduce RAM requirements. The results suggest that optimizing for RAM usage needs new combinations of optimizations, different from those suitable for optimizing for speed or code size.

Impact of Optimizations on RAM Usage

Most current compilers optimize for execution speed and/or code size. While reducing code size was an important issue in the early years of compiler technology, with the introduction of microcomputers, focus in research of compiler optimizations has mostly been on execution speed [141]. This is reasonable, as most compiler research has been targeted at personal computers or larger systems where code size rarely matters. However, larger code can cause slower execution, e.g. due to cache misses, in many cases. In fact, many optimizations to improve execution speed also reduce code size.

Optimizing code size and execution speed are the classical objectives of optimization. For the small embedded systems addressed in this thesis, however, it has been shown in the previous chapters that there is an additional objective, namely RAM size. There are obvious reasons why RAM usage has not been addressed in research of compiler optimizations. Typical optimizations will change the use of RAM only slightly by introducing or removing (usually scalar) variables. On any but the smallest systems, these effects can be neglected as RAM usage is dominated either by code size (if code is loaded into RAM as it is done in most larger systems) or user data structures (which are typically not modified by common optimizations). On a system with only 2KB of RAM, however, a few bytes more can make a difference.

3.1 Why Optimizing?

Quality of generated code has been an issue since the early days of compiler construction. It was not long before it was proposed to carry out optimizing transformations automatically (see [108]). In the 1960s, smaller optimizations were included in compilers [152] and the first heavily optimizing compilers were produced, e.g. the Fortran H compiler (see [97]). Since then, much research into compiler optimizations has been carried out (e.g. [4, 83]). Today, there are still many conferences dealing with compiler optimizations, e.g. the "International symposium on Code generation and optimization" or, in the area of embedded systems, "Languages, Compilers and Tools for Embedded Systems" (LCTES), the "International conference on Compilers, architecture, and synthesis for embedded systems" (CASES) or the "International Workshop on Software and Compilers for Embedded Systems" (SCOPES). Among the targets of research are better support for new architectures and languages, or increasing the scope of optimizations.

As the computing power of microprocessors has been increasing very fast over the years, the usefulness of optimizing compilers is questioned regularly. At first sight, it is understandable as optimizing compilers inherently incorporate a number of disadvantages compared to their non-optimizing counterparts.

3.1.1 Disadvantages of Optimizing Compilers

The first apparent disadvantage optimizing compilers have are probably the much larger compilation times and memory requirements. While non-optimizing compilers will on average usually have linear complexity in time and space, optimizing compilers rarely do better than quadratic — often with even higher theoretical worst-case complexity in time. Faster compilation times are only possible for some languages and only with restrictions to the optimizations performed (see [23]).

Maybe the second most obvious problem for the user of the compiler will be the restrictions when debugging. Most compiler/debugger combinations offer either only very limited optimizations when debugging information is selected, or the debugger may display very confusing results.

One problem here is that most standard formats for debug information have been designed with simple compilers in mind. For example, they only support very simple expressions for locating variable values (e.g. constant offset to a frame-pointer register). In an optimizing compiler, however, a variable might be in a register at some part of the code, another register in other parts and reachable via changing offsets to a stack-pointer at yet another part of the code.

These shortcomings can be eliminated by sophisticated formats for debug information and (this is not always the case) debuggers that support it. For example, the DWARF2 format (see [140]) offers a reasonable variety of possibilities that support debugging code produced by optimizing compilers. Unfortunately, it is a pretty complex specification and most debuggers seem to support only a part of the functionality that is specified.

So, while some problems with debugging could be solved, there are others that simply can not be eliminated with the user interface of current source-level debuggers. For example, if variables or assignments have been removed or combined, there is no way the debugger can show the right value of a variable at some locations. In the following example, an optimizing compiler might just increment the parameter `x` (e.g. if it is passed in a register) and return it.

```
int f(int x)
{
  int y = x + 1;
  return y;
}
```

As a result, neither of the two variables `x` and `y` would be displayed correctly over the function body. There are, of course, special cases where it would be possible to give the debugger information that would enable it to calculate the correct value of a variable that has been eliminated. For example, the value of an induction variable that has been eliminated by strength-reduction (see section 3.3.7) could be calculated from a pointer variable that has been introduced by the optimizer. In the general case, however, this is not possible.

Also, single stepping through the source code or setting break-points on source lines presents problems. Often machine instructions correspond to several source lines after instruction combining or machine instructions corresponding to different source lines are intermixed by instruction scheduling. Many optimizations destroy the one-to-one mapping of sequences of machine instructions to source lines. If this mapping is destroyed, neither single-stepping nor breaking at certain source-lines can work as expected.

The previous paragraphs explained some problems for the users of optimizing compilers. Another group of problems affect the developers writing the compilers and therefore development cost and stability of compilers.

Optimizing compilers are much more complicated and error-prone than simple non-optimizing compilers. They use a wide range of different algorithms (e.g. data-flow algorithms, pattern-matching, dynamic-programming, sorting, searching, etc.) and complex data-structures (e.g. trees, directed acyclic graphs, control-flow graphs, bit-vectors, hash-tables, lists, etc.).

Often different algorithms work sequentially or even interconnected on the same data-structures and pass information to each other. Therefore, bugs in an optimizing compiler may show up in a different part of the compiler (e.g. a wrongly propagated constant may be the result of corrupt data-flow information which may, in turn, be the result of a bug in the alias-analysis algorithm).

Additionally, the global analysis in optimizing compilers often causes bugs to appear at different parts of the translated program. For example, corrupt data-flow analysis at the end of a loop may be the cause for an incorrectly translated statement at the head of the loop. Furthermore, it is much harder to produce a small code snippet that reproduces a bug. Adding statements (like printing debugging output) as well as removing parts of code (to reduce the code to a minimum which is easier to check) may cause changes to the generated code at locations far away.

To sum it up, the steps in isolating and fixing an optimizer bug are usually much more complicated and not at all straight-forward like they are in a simple non-optimizing compiler.

3.1.2 Advantages of Optimizing Compilers

So now we have seen a series of disadvantages of optimizing compilers. Their main advantages obviously are speed and/or size of the generated code. Therefore, the justification of sophisticated optimizations depends on the importance of code quality. If the additional efficiency delivered through extensive optimization is not needed, optimizing compilers should better be avoided.

The major point to support this criticism is the fast increase of computing power. As a counterpart to "Moore's Law" which basically states that computing power doubles every 18 *months* due to advances in hardware technology, Proebsting postulates that improvements in compiler optimizations double computing power every 18 *years* [123]. While this is of course more a provocative anecdote than hard research, the basic tendency surely is correct. The impact of compiler optimizations has not nearly increased computing power as much as new hardware did. In fact, new optimizations often tend to improve the generated code less than previous optimizations did. If current compilers already generate code that is close to optimal then there is little room for improvement. So it is clear that new compiler optimizations have not and will not cause exponential improvements like hardware technology has done in the past.

On first sight, for many of today's applications, advanced optimizations indeed seem unnecessary. This is especially true if you are mostly familiar with typical personal computers and the most common applications like word processors, spreadsheets, etc. In this area it can be observed that even current low-end personal computers are powerful enough to deal with such applications. More so, the number of bugs observed in such applications strengthens the assumption that the ease of use of development tools and the speed of the development cycle is much more beneficial than optimization algorithms.

On the other hand, one has to keep in mind that there are other applications and other target domains that may have completely different requirements. Even on standard personal computers there are highly speed-critical applications like games or multimedia programs. Similarly, scientific number crunchers always can make use of as much computing power as is given to them (see [18]).

Of course the system-on-chip microcontrollers which are the main subject in this thesis also fall into this category. Here, larger memory footprints almost directly influence hardware costs by requiring larger on-chip memories or perhaps even off-chip memory including glue logic.

Similarly, slower code often adds to costs. If the code is too slow to guarantee the required real-time behaviour, a faster chip may have to be selected. This may become especially problematic as higher clock-speeds usually imply more power consumption and a higher sensitivity to electro-magnetic interference.

Even if the chosen chip is able to execute the non-optimized code fast enough to satisfy all real-time constraints, the optimized code may still need less power as it finishes the work in less time and therefore spends more time in idle or power-saving mode. In many embedded applications, power-consumption is a serious issue and therefore optimizations are desired.

As the next bigger chip may cost a few Euros more, the non-optimized code may cost millions of Euros in high-volume production if it forces the switch to a bigger chip. Or, equivalently, optimizations may save millions if they enable the application to use a cheaper microcontroller. As many production compilers are usually used to compile at least a few of such applications, this gives an estimation of how much effort might be justified to be put into an optimizing compiler.

So we have seen that compiler optimizations are still a topic of significant interest and of much practical use. In fact, the question of the purpose of optimizing compilers has been put up many years before when computing power was some orders of magnitude smaller than what is now considered suitable [141]. Actually, the topic is brought up regularly, see, for example, [59]. All these discussions have not changed the fact that these days the average level of available optimizations in production compilers has significantly and constantly increased over the years.

3.2 Related Work

Optimizing for code space has been an important topic in early compilers (see e.g. [97]) and has become an active topic of research again, especially for embedded systems. New approaches are, for example, presented in [10] or [44]. The problem of "over optimization", i.e. overly aggressive application of transformations resulting in degraded performance is mentioned, for example, in [11]. Methods to reduce register pressure are presented in [84]. However, these papers only consider the impact on execution speed. Similarly, many optimizations to improve cache performance have been discussed (see, for example, [101, 154]). A framework to examine the impact of optimizations on cache behaviour of embedded applications is presented in [156]. Reducing the power consumption of embedded systems by compiler optimizations has also become an active topic of research (see, for example, [155]). However, optimizing for power consumption is almost always identical to optimizing for execution speed as is shown in [35]. An overview of current works dealing with memory optimization techniques for embedded systems is presented in [121].

3.3 Discussion of selected Optimizations

Below, a selected list of probably the most used and frequently applied optimizations is reviewed regarding their applicability for small embedded systems, especially their impact on RAM usage. Code examples are usually written in C as the C language allows the effect of most optimizations to be expressed in source code. Note however, that some of these cases would rarely occur in code written by the programmer, but rather in intermediate code, possibly after other optimizations have already been performed.

After introducing each optimization, some comments on their implementation in vbcc will be given. The impact of several optimizations on RAM usage is measured using the 21 parts of the C torture test (`cq.c`) that is described in section 4.2.3. When evaluating a certain optimization, the corresponding transformation has been turned off and the results are compared to those obtained with vbcc default settings (marked "**Ref.**") on highest optimization level (unless mentioned otherwise). These default settings activate all optimizations apart from the most aggressive version of loop unrolling (see section 3.3.8). Therefore, the reference is usually with the optimization *on*. While this may be confusing at first, it allows the use of the same reference for all tests.

The first table (3.1) includes all test cases (with the RAM usage of each test case in bytes computed by the compiler), while the following tables contain only those entries where the settings that have been tested make a difference (this has to be considered when reading the total values at the bottom of each table). The PowerPC was chosen as target architecture because it is best supported by vbcc (as the optimizations discussed here are done in a machine-independant way, similar results can be expected for other architectures). There are very complex interactions between all the optimizations performed, and therefore side effects caused by other optimizations distort the results somewhat. Nevertheless, this approach seems more realistic than turning off all optimizations as a base for reference.

3.3.1 Flow Optimizations

The basic data-structure most optimization algorithms operate on, is the control-flow graph. Its nodes represent pieces of code (usually basic blocks of intermediate code, sometimes machine instructions, etc.). The edges are possible flows of control.

Several optimizations working mainly on the control flow can be easily implemented and are done by many compilers. For example, code which is unreachable will be removed and branches to other branches or branches around branches will be simplified.

For example, the following code

```
void f(int x, int y)
{
  if(x > y)
    goto label1;
  q();
label1:
  goto label2;
  r();
label2:
}
```

can be optimized to:

```
void f(int x, int y)
{
  if(x <= y)
    q();
}
```

Many different control flow optimizations have been implemented for many years, e.g. in [97]. Control flow analysis methods have been discussed for example in [5].

These optimizations usually eliminate unnecessary branch instructions and often enable additional optimizations. Therefore, they generally increase speed and reduce the size of the generated code. RAM space is not affected. That is why this kind of optimizations is suitable and recommended for small embedded systems. There are probably few or no specific considerations necessary for such systems.

vbcc always performs a series of control-flow optimizations when building a control flow graph. Also, several such transformations are spread over several optimization phases and are hard to remove. Therefore, no results are provided for the effect of those optimizations in vbcc.

3.3.2 Dead Assignment Elimination

If a variable is assigned a value which is never used (either because it is overwritten or its lifetime ends), the assignment will be removed by this optimization. Although source code usually contains few dead assignments, this optimization is crucial to remove code which has become dead due to other optimizations.

For example, the following code

```
int x;

void f()
{
  int y;
  x = 1;
  y = 2;
  x = 3;
}
```

can be optimized to:

```
int x;

void f()
{
  x = 3;
}
```

This optimization has been performed by [97], other early approaches can be found in [83] or [4]. An extension that can also remove partially dead code is presented in [85]. Dead assignment elimination is crucial to remove "garbage" produced by other optimizations. As it only removes code, it can have no negative effect on speed or size of the generated code (academic examples aside). It should be done by a compiler for small systems and can be used without specific considerations.

By default, vbcc performs standard elimination of dead assignments using data-flow information on live variables. Table 3.1 shows how RAM usage is affected when this optimization is turned off. As expected, the stack usage grows heavily, because many temporary variables introduced by other optimizations are left on the stack. The improvement in function `s243` is caused by prevention of other optimizations that increase RAM usage in this case.

Table 3.1: Dead Assignment Elimination Results

Func	Ref.	NDAE	%
s22	0	32	n/a
s241	560	592	106
s243	320	304	95
s244	80	128	160
s25	0	64	n/a
s26	0	16	n/a
s4	16	80	500
s61	48	144	300
s626	144	464	322
s71	0	80	n/a
s72	0	304	n/a
s757	160	384	240
s7813	0	48	n/a
s714	0	3200	n/a
s715	0	48	n/a
s81	304	784	258
s84	288	336	117
s85	288	448	156
s86	32	208	650
s88	32	32	100
s9	0	16	n/a
total	**2272**	**7712**	**339**

Ref.: Global dead assignment elimination
NDAE: No dead assignment elimination

3.3.3 Constant Propagation

If a variable is known to have a constant value (this includes, for example, addresses of static objects) at some location, this optimization will replace the variable by the constant.

For example, the following code

```
int f()
{
  int x;
  x = 1;
  return x;
}
```

can be optimized to:

```
int f()
{
  return 1;
}
```

The scope of this optimization varies from simple intra-expression to whole-program inter-procedural analysis. Global constant propagation is presented, for example, in [83]. Improvements can be found, for example, in [147], [28], or [29]. The optimization does not introduce new temporaries or new code. The only situation where it has a negative effect might be if using a constant is more expensive than using the variable (e.g. many architectures have to use a load-from-memory instruction to load a floating-point constant whereas a floating-point value in a variable can easily be held in a machine register).

This optimization should be followed by constant folding and often allows additional optimizations to be made. It is very profitable and should be used in small embedded systems. Care has to be taken only for "large" constants as mentioned above (this is not specific to small systems, however). Including large constants as candidates for register-allocation prior to code-generation is one possible solution for this problem.

As the live-range of a variable might be reduced or a variable might be eliminated altogether, constant propagation can have a positive effect on RAM usage.

By default, vbcc performs global constant propagation. All constants, including addresses of static objects are propagated. However, constants may be assigned to registers prior to code generation (see section 3.3.10). Table 3.2 shows the impact on RAM usage if constant propagation is only done locally or disabled completely. The results confirm the positive effect on RAM usage.

3.3.4 Common Subexpression Elimination

Common subexpression elimination (CSE) is perhaps one of the most typical optimizations and tries to eliminate re-computation of expressions that have already been calculated before. A simple example would be the transformation of

```
void f(int x, int y)
{
  q(x * y, x * y);
}
```

into

Table 3.2: Constant Propagation Results

Func	Ref.	LCP	%	NCP	%
s243	320	320	100	288	90
s244	80	80	100	96	120
s25	0	0	n/a	32	n/a
s4	16	32	200	48	300
s61	48	48	100	64	133
s626	144	160	111	128	89
s71	0	0	n/a	64	n/a
s72	0	48	n/a	48	n/a
s757	160	160	100	176	110
s7813	0	32	n/a	80	n/a
s714	0	0	n/a	32	n/a
s81	304	320	105	304	100
s84	288	416	144	256	89
s85	288	288	100	336	117
s86	32	80	250	96	300
s9	0	0	n/a	16	n/a
total	**2272**	**2576**	**113**	**2656**	**117**

Ref.: Global constant propagation
LCP: Local constant propagation
NCP: No constant propagation

```
void f(int x, int y)
{
  int tmp;

  tmp = x * y;
  q(tmp, tmp);
}
```

Many common subexpressions are less obvious like in:

```
extern int a[];

void f(int i)
{
  return a[i + 1] + a[i + 2];
}
```

In this example, the calculation of `i * sizeof(int)` (which is hidden in C, but necessary on machine level) is a common subexpression after re-arranging the array access.

Also, in the examples above, the common subexpressions have been within a statement. This is the simplest case and the easiest one to detect by a compiler. A somewhat more powerful approach is to look for common subexpressions within a basic block. This is still not much more difficult than within a statement. CSE has been performed in [97]. Algorithms for global CSE are presented in [39] or [83]. Further improvements have been made, e.g. partial redundancy elimination extends CSE and loop invariant

code motion. This optimization was first presented by [104] and is still a topic of active research (see e.g. [26, 27]).

There may also be several computations of a common subexpression, i.e.

```
void f(int x, int y, int a)
{
  if(a){
    x++;
    q(x * y);
  }else{
    y++;
    r(x * y);
  }
  s(x * y);
}
```

has to be optimized into:

```
void f(int x, int y, int a)
{
  int tmp;
  if(a){
    x++;
    tmp = x * y;
    q(tmp);
  }else{
    y++;
    tmp = x * y;
    r(tmp);
  }
  s(tmp);
}
```

Whether the optimization is beneficial here is much less obvious. There are two additional assignments to the `tmp` variable. If it can be placed in a register it may not cost any additional machine instructions (i.e. the value of `x * y` can already be calculated in this register).

If, however, there is no free register for the variable, one multiplication has been replaced by two moves which may or may not be better depending on the target machine. The number of locations the expression is computed at influences this decision, as well as the number of eliminated re-computations and the complexity of the operation. Also, sometimes the computations can be hoisted (in the example above, this would be possible if the `x++` and `y++` statements were missing).

Unfortunately, much of this is not available when the decision is made as to whether the transformation should be performed or not. For example, one application of the optimization usually eliminates only an expression as contained in one instruction of the intermediate code. Therefore, an expression like `x * y * z` would be eliminated in two steps as the intermediate code probably looks like `t1 := x * y; t2 := t1 * z`. Similarly, common subexpression elimination often exhibits additional opportunities for other optimizations. This is, however, also usually known only after common subexpression elimination has indeed been performed.

By default, vbcc performs global common subexpression elimination. Table 3.3 shows the impact on RAM usage if this optimization is only done locally or disabled

completely. The results suggest that common subexpression elimination should **not** be performed as long as RAM usage is the only concern. While this was expected, the huge RAM usage in test case `s626` is an unusual case. An array is filled with values and afterwards read again (in a loop that has been unrolled). Common subexpression elimination treats the address of each array element as common subexpression and keeps it in a register or on the stack. Re-materialization or more conservative application of common subexpression elimination could reduce RAM usage for this test case without disabling the optimization.

Table 3.3: Common Subexpression Elimination Results

Func	Ref.	LCSE	%	NCSE	%
s243	320	288	90	288	90
s626	144	144	100	48	33
s757	160	176	110	160	100
s81	304	304	100	256	84
s86	32	0	0	0	0
total	**2272**	**2224**	**98**	**2064**	**91**

Ref.: Global common subexpression elimination
LCSE: Local common subexpression elimination
NCSE: No common subexpression elimination

3.3.5 Copy Propagation

If a variable is assigned to another one, this optimization will try to use the original one as long as it is not modified. Sometimes all uses of the second variable can be replaced and it can be removed completely (by dead assignment elimination, see section 3.3.2).

For example, the following code

```
int y;

int f()
{
  int x;
  x = y;
  return x;
}
```

can be optimized to:

```
int y;

int f()
{
  return y;
}
```

Often, it enables additional optimizations and works together smoothly with e.g. common subexpression elimination. The optimization can either be done fast on basic blocks or over entire procedures requiring data-flow analysis. An early version of this

optimization is implemented in [97], advanced versions are, for example, presented in [143].

No new variables are introduced by this optimization and no additional code is produced — only new opportunities for optimization may be created. Therefore, this optimization is crucial and not critical for small embedded systems. The global version of this optimization seems preferable.

By default, vbcc performs global copy propagation. Table 3.4 shows the impact on RAM usage if only local propagation is performed or if the transformation is disabled completely. The results suggest that copy propagation can reduce RAM requirements. It must be noted, however, that copy propagation works very closely with CSE and exposes more common subexpressions. As CSE tends to increase RAM usage, some of the benefits of copy propagation may be overshadowed when using CSE. This can be seen in test case `s626` which is the one that was heavily influenced by CSE. If no copy propagation is performed, CSE will also not work in that case. This explains why the results without any copy propagation lie inbetween local and global copy propagation.

Table 3.4: Copy Propagation Results

Func	Ref.	LCOP	%	NCOP	%
s243	320	320	100	288	90
s25	0	0	n/a	16	n/a
s4	16	32	200	32	200
s61	48	64	133	64	133
s626	144	144	100	64	44
s71	0	0	n/a	16	n/a
s81	304	352	116	336	111
s84	288	288	100	240	83
s85	288	336	117	336	117
s86	32	80	250	80	250
s88	32	32	100	32	100
total	**2272**	**2448**	**108**	**2304**	**101**

Ref.: Global copy propagation
LCOP: Local copy propagation
NCOP: No copy propagation

3.3.6 Loop-Invariant Code Motion

If the operands of a computation within a loop will not change during iterations, loop-invariant code motion will move the computation outside of the loop, so that it will be computed only once rather than recomputed for every iteration.

For example, the following code

```
void f(int x, int y)
{
  int i;

  for (i = 0; i < 100; i++)
    q(x * y);
}
```

can be optimized to:

```
void f(int x, int y)
{
  int i, tmp = x * y;

  for (i = 0; i < 100; i++)
    q(tmp);
}
```

Simple versions of this optimizations have been implemented in [152] and [97]. In recent research, code motion is often done in the context of partial redundancy elimination (see [104, 26, 27]). While it can be argued that programmers can do this optimization on source-level, there are still often opportunities for loop-invariant code motion. First, some cases are easily missed or make the source code less readable. When optimizing on source-level, the following example

```
void f(int z)
{
  int x, y;
  for(x = 0; x < 100; x++)
    for(y = 0; y < 100; y++)
      q(x * x + y * y + z * z);
}
```

has to be rewritten to:

```
void f(int z)
{
  int t1 = z * z;
  for(x = 0; i < 100; x++){
    int t2 = x * x:
    for(y = 0; y < 100; y++)
      q(t2 + y * y + t1);
  }
}
```

Furthermore, expressions involving array accesses sometimes contain loop-invariant code that can not (or only with very "ugly" code) be moved out of the loop at source-level.

For most cases, this transformation will improve the speed of the generated code which makes it a promising optimization if speed is a major issue. However, it introduces a new temporary variable which is live across the entire loop. Unless other variables can be eliminated (e.g. if variables used in the loop-invariant expression are not used elsewhere in the loop), it can increase register pressure and/or increase RAM requirements.

Table 3.5 shows no significant impact on RAM usage when vbcc performs loop invariant code motion. Therefore, this optimization should be an option for small embedded systems if speed is of interest. As it may increase code-size as well as RAM requirements, it should be used with care and perhaps be turned off for small systems where ROM/RAM size is a major consideration.

Table 3.5: Loop Invariant Code Motion Results

Func	Ref.	NINV	%
s81	304	320	105
total	**2272**	**2288**	**101**

Ref.: Loop invariant code motion
NINV: No loop invariant code motion

3.3.7 Strength-Reduction

This is an optimization applied to loops in order to replace more costly operations (usually multiplications) by cheaper ones (typically additions). Linear functions of an induction variable (a variable which is changed by a loop-invariant value in every iteration) will be replaced by new induction variables. If possible, the original induction variable will be eliminated.

As array accesses are actually composed of multiplications and additions, they often benefit significantly by this optimization.

For example, the following code

```
void f(int *p)
{
  int i;

  for (i = 0; i < 100; i++)
    p[i] = i;
}
```

can be optimized to:

```
void f(int *p)
{
  int i;

  for (i = 0; i < 100; i++)
    *p++ = i;
}
```

Data-flow analysis is needed for reasonable implementation of this transformation. Early implementations are described in [152] and [97]. A more detailed presentation can be found in [40]. For more recent research results, see, for example, [45].

Some other optimizations are related to this one and may allow to eliminate the induction variable in some cases. For example, if an induction variable is only used to determine the number of iterations through the loop, it can be removed. Instead, a new variable will be created which counts down to zero. This is generally faster and often enables use of special decrement-and-branch or decrement-and-compare instructions.

The following code

```
void f(int n)
{
  int i;

  for (i = 0; i < n; i++)
    puts("hello");
}
```

can be optimized to:

```
void f(int n)
{
  int tmp;

  for(tmp = n; tmp > 0; tmp--)
    puts("hello");

}
```

Another possibility to eliminate the induction variable is to replace the original loop-test by a comparison of the new induction variable against the linear function of the original loop boundary, e.g. the following code

```
extern int a[];

void f()
{
  int i;

  for (i = 0; i < 100; i++)
    a[i] = 0;
}
```

can be transformed to:

```
extern int a[];

void f()
{
  int *p;
  for (p = &a[0]; p < &a[100]; )
    *p++ = 0;
}
```

In many cases, speed and size of the generated code will improve after applying strength-reduction. However, a new variable will be introduced and live across the entire loop. Therefore, RAM requirements could increase as well as code-size. Table 3.6 shows the impact of strength reduction on RAM usage in vbcc. The reference settings introduce new variables for all linear functions of induction variables. These base values are compared with the results of only reducing multiplications and of completely disabling loop strength reduction. The best results in this test have been obtained with reducing only multiplications.

Many microcontrollers do not offer fast multiplication. On such machines, strength reduction should at least be an option. On the other hand, some small devices are better at handling indices rather than pointers. For example, an 8bit microcontroller may better be able to add an 8bit index to an invariant pointer in a 16bit index-register rather than performing a 16bit addition to the index-register. Together with the considerations about adding a temporary variable (like already mentioned for previous optimizations), this optimization should be used with care for small embedded devices.

Table 3.6: Strength Reduction Results

Func	Ref.	SRMUL	%	NSR	%
s84	288	208	72	208	72
s86	32	32	100	48	150
total	**2272**	**2192**	**96**	**2208**	**97**

Ref.: Strength reduction for addition and multiplication
SRMUL: Strength reduction for multiplication only
NSR: No strength reduction

3.3.8 Loop Unrolling

The loop overhead (such as testing the loop-condition and conditional branching) can be reduced by replicating the loop body and reducing the number of iterations. Also, additional optimizations between different iterations of the loop will often be enabled by creating larger basic blocks. However, code-size can increase significantly. Loop unrolling is described, for example, in [50] or [106], more recent research can be found in [49] or [126].

If the number of iterations is constant and the size of the loop body is small, a loop can be unrolled completely:

```
void f()
{
  int i;

  for (i = 0; i < 4; i++)
    q(i);
}
```

can be optimized to:

```
void f()
{
  q(0);
  q(1);
  q(2);
  q(3);
}
```

The loop can still be unrolled several times if the number of iterations is constant but not a multiple of the number of replications. The remaining iterations must be unrolled separately (or a second loop must be generated).

For example, the following code

```
void f()
{
  int i;

  for (i = 0; i < 102; i++)
    q(i);
}
```

can be optimized to:

```
void f()
{
  int i;
  q(0);
  q(1);
  for(i = 2; i < 102;){
    q(i++);
    q(i++);
    q(i++);
    q(i++);
  }
}
```

If the number of iterations is not constant, but can be calculated before entering the loop, unrolling is still possible in some cases:

```
void f(int n)
{
  int i;

  for (i = 0; i < n; i++)
    q(i);
}
```

can be optimized to:

```
void f(int n)
{
  int i, tmp;

  i = 0;
  tmp = n & 3;
  switch(tmp){
  case 3:
    q(i++);
  case 2:
    q(i++);
  case 1:
    q(i++);
  }
  while(i < n){
    q(i++);
```

```
    q(i++);
    q(i++);
    q(i++);
  }
}
```

In most cases, loop unrolling trades code size for speed and some compilers, e.g. for workstations [19], do it very aggressively. When code size is an issue, it usually will not be used. While this makes sense in most cases, there still are loops which actually may get smaller when they are unrolled completely. If the loop-body is small and the loop is executed only a few times (consider, for example, a 3D vector-addition) unrolling may produce faster and smaller code.

With traditional loop unrolling, no additional memory requirements should be produced. Some compilers may choose to introduce new variables for the different unrolled loop iterations to get better instruction level parallelism, but this borders software pipelining (see section 3.3.11). However, loop unrolling may exhibit some opportunities for other optimizations that may, in turn, increase RAM usage. vbcc can completely or partially unroll loops with a constant number of iterations (which is done by default) as well as loops where the number of iterations has to be computed at run time. A command line parameter specifies the maximum size of the unrolled loop body in terms of intermediate instructions. It controls the number of iterations that are unrolled depending on the size of the loop. Table 3.7 shows the effect on RAM usage for the default behaviour, no loop unrolling and unrolling of all loops (where the number of iterations has to be computed at run time).

Loop-unrolling is relatively important if speed is a major consideration. If code size is more important, only the first form (completely unrolling the loop) is promising in some cases.

Table 3.7: Loop Unrolling Results

Func	**Ref.**	**NUR**	**%**	**URA**	**%**
s243	320	288	90	320	100
s25	0	16	n/a	0	n/a
s4	16	48	300	16	100
s626	144	80	56	144	100
s72	0	0	n/a	16	n/a
s757	160	160	100	176	110
s84	288	240	83	288	100
s86	32	64	200	32	100
total	**2272**	**2208**	**97**	**2304**	**101**

Ref.: Unrolling of loops with a constant number of iterations
NUR: No loop unrolling
URA: Unrolling of all loops where possible

3.3.9 Function Inlining

To reduce overhead, a function call can be expanded inline. Passing parameters can be optimized as the arguments can be directly accessed by the inlined function. Also, fur-

ther optimizations are enabled, e.g. constant arguments can be propagated or common subexpressions between the caller and the callee can be eliminated.

For example, the following code

```
int f(int n)
{
  return q(&n,1);
}

void q(int *x, int y)
{
  if(y > 0)
    *x = *x + y;
  else
    abort();
}
```

can be optimized to:

```
int f(int n)
{
  return n + 1;
}

void q(int *x, int y)
{
  if(y > 0)
    *x = *x + y;
  else
    abort();
}
```

Early descriptions can be found, for example, in [14] or [67]. More recent research can be found, for example, in [30], [91] and [48]. Function inlining usually improves speed of the generated code. Obviously, too much inlining can increase code size significantly (see [91]). This is why it is often seen as an optimization not useful for small embedded systems. Careful inlining, however, can actually decrease code size as the body of small functions is sometimes smaller than the function call overhead imposed by the application binary interface (ABI) which is needed to call them.

A further important point is the effect on RAM requirements. After inlining, less RAM may be needed (see [124]). Many ABIs require that arguments are pushed on the stack, a certain stack alignment etc. At the very least, the return address has to be stored somewhere. Therefore, function inlining is a very promising optimization if speed or RAM requirements are major issues. Even if code size is top priority, careful application of this transformation can be an option.

Some further considerations about the scope of this optimization will be made below (see section 3.3.14). Inlining in vbcc is controlled via two options that specify the maximum size of functions to inline as well as the maximum depth of inlining (i.e. how many passes of inlining will be done). Table 3.8 shows the impact on RAM usage of the default setting, disabled inlining, and extensive inlining of all possible functions. The results suggest that reasonable use of inlining can reduce memory requirements, but overly aggressive inlining can cause side effects that eat up the benefits (which is probably a side effect of other optimizations).

Table 3.8: Function Inlining Results

Func	Ref.	NIL	%	ILA	%
s241	560	544	97	560	100
s243	320	272	85	320	100
s25	0	16	n/a	0	n/a
s4	16	48	300	16	100
s71	0	32	n/a	0	n/a
s72	0	16	n/a	0	n/a
s757	160	176	110	160	100
s7813	0	16	n/a	0	n/a
s715	0	16	n/a	0	n/a
s81	304	320	105	608	200
s84	288	304	106	288	100
s85	288	304	106	288	100
s86	32	48	150	32	100
total	**2272**	**2416**	**106**	**2576**	**113**

Ref.: Normal function inlining
NIL: No function inlining
ILA: Inlining of all functions

3.3.10 Register Allocation

Mapping the most used variables to machine registers and reusing these registers efficiently surely is among the most profitable optimizations. Various algorithms have been proposed. Theoretical aspects of local register allocation are addressed in [71] and [98]. Actual, already rather sophisticated, implementations are described in [152] and [97]. Use counters are described in [61]. A graph coloring register allocator is presented in [34], possible improvements are discussed in [25]. An alternative approach is presented in [38]. Register allocation is still actively researched. For example, the scope is extended in [146], [58] or [17] (which is presented in section 5.3.3). Also, register allocation especially for embedded systems is researched, e.g. in [36] and [127].

Good register allocation manages to keep the most used variables in registers, reduces the number of stack slots used, and minimizes moves between registers by calculating values in the machine register they are needed in (e.g. for argument passing or special-purpose machine registers).

Therefore, good register allocation benefits speed, code size, and RAM requirements, and is crucial for small embedded systems (as for pretty much any system). The more difficult question, however, is what algorithm for register allocation delivers best results for a given system. Perfect register allocation is obviously at least as hard as finding a k-coloring for a graph, which is NP-complete. For example, graph coloring algorithms [34, 25] have been shown to produce very good results for RISC-type machines, but they are not easily usable for architectures with a very small non-orthogonal register set (see [73]).

vbcc does not use a graph coloring register allocation, but uses a combined approach of local allocation and hierarchical live range splitting. Cost savings are computed, guided by cost functions provided by the backend. Then the most promising variables will be assigned to registers, first for the entire function, then for loops, starting with

the outermost loops. This algorithm is similar to [38] and [97]. Its goal is to make best use of non-orthogonal register sets in innermost loops. However, this can cause shuffling of variables to different registers between loops. In general, this approach is tailored to speed rather than size. Whether a graph coloring register allocator can be used in the vbcc framework and can produce good results also for non-orthogonal architectures, is examined in [73]. The results show that graph-coloring can be added to vbcc and yields similar results as the original register allocator of vbcc for RISC architectures. To obtain good results also for non-orthogonal instruction sets, significant modifications to the classical graph-coloring register-allcoation algorithm are needed.

vbcc assigns constants to registers if that improves speed on the target, but this can increase RAM usage. Similarly, vbcc is able to cache global variables or memory references in a register across a loop if it can detect that they are not accessed through aliases within the loop. This optimization can only be applied in some cases and tends to increase RAM usage. Furthermore, vbcc takes register usage of called functions into account to perform inter-procedural register-allocation similar to [37]. A new extension to register allocation, specially tailored to small embedded systems using static operating systems will be presented later in this thesis (see section 5.3.3).

Table 3.9 shows the results of register allocation in vbcc with default settings, with register allocation turned off for constants, and with only local register allocation. Apparently, local register allocation (i.e. only within basic blocks) is not sufficient for good results, and the allocation of constants to registers obviously has a bad effect on RAM usage. As vbcc performs aggressive inlining with default settings, table 3.10 shows the positive effect of inter-procedural register allocation with inlining disabled.

3.3.11 Instruction Scheduling

Most modern processors, including very small devices, use internal pipelines to be able to increase the throughput. There are, however, often certain code-patterns that cause pipeline-stalls. For example, an instruction depending on register r, directly after an instruction that loads register r from memory, might cause a pipeline-stall.

The exact rules to avoid pipeline-stalls are highly processor-specific (not just architecture-specific) and can be highly complex. Sometimes there are dependencies in the code that make it impossible to avoid pipeline-interlocks. Often, however, reordering of the code allows at least some reduction of stalls and increases the throughput. Many small microcontrollers are not very sensitive to pipeline-hazards. On processors with long pipelines, however, these effects can be significant. For example, on an Alpha 21064 processor, the following code

```
ldq  r0,a   /* load a   */
stq  r0,b   /* copy to b */
ldq  r0,c   /* load c   */
stq  r0,d   /* copy to d */
```

will cause pipeline stalls and therefore run much slower than this version:

```
ldq  r0,a   /* load a   */
ldq  r1,c   /* load c   */
stq  r0,b   /* copy to b */
stq  r1,d   /* copy to d */
```

There are various algorithms to perform instruction scheduling. The simplest case is probably just reordering machine instructions in a basic block. More sophisticated approaches move instructions across basic block boundaries and will not only reorder

Table 3.9: Register Allocation Results

Func	Ref.	NCRA	%	LRA	%
s22	0	0	n/a	32	n/a
s241	560	512	91	512	91
s244	80	80	100	112	140
s25	0	0	n/a	32	n/a
s26	0	0	n/a	32	n/a
s4	16	16	100	48	300
s61	48	48	100	64	133
s71	0	0	n/a	32	n/a
s72	0	0	n/a	32	n/a
s757	160	160	100	128	80
s7813	0	0	n/a	32	n/a
s714	0	0	n/a	32	n/a
s715	0	0	n/a	32	n/a
s81	304	304	100	320	105
s84	288	288	100	352	122
s85	288	288	100	304	106
s86	32	32	100	48	150
s88	32	32	100	48	150
s9	0	0	n/a	32	n/a
total	**2272**	**2224**	**98**	**2688**	**118**

Ref.: Global register allocation
NCRA: No register allocation for constants
LRA: Local register allocation only

Table 3.10: Inter-Procedural Register Allocation Results

Func	NIL	NIPRA	%
s243	272	288	106
s244	80	96	120
s25	16	32	200
s626	144	160	111
s7813	16	32	200
s715	16	32	200
total	**2416**	**2512**	**104**

NIL: No function inlining
NIPRA: No function inlining and no inter-procedural register allocation

code but also modify the code. See [69] or [62] for examples of scheduling algorithms. New research with the goal of power saving can be found, for example, in [90]. Code that reuses a register might be harder to schedule than code that uses different register for subsequent computations, therefore a single variable may be expanded to several different variables (see [99]).

Further variants are, for example, software pipelining (interleaving of loop-iterations, see [89]) or branch scheduling (filling useful instructions into branch delay slots or between a compare and the corresponding branch instruction, see [100]). Those scheduling techniques are often used together with function inlining and loop unrolling. Also, the benefits are larger on processors with long pipelines and instruction-level parallelism. Small embedded microcontrollers usually are not that dependant on advanced scheduling techniques.

The scheduling algorithms that just reorder machine code only have a positive effect on execution speed (although somehow limited) but no negative side-effects. The more advanced techniques may increase code size or increase register pressure (by using more registers) and may therefore increase RAM and ROM usage. vbcc only performs reordering of machine instructions within basic blocks after code generation. While this is not as effective as more sophisticated algorithms, it is guaranteed never to affect RAM usage (and therefore no table is provided).

3.3.12 Alias Analysis

Many optimizations can only be done if it is known that two expressions are not aliased, i.e. they do not refer to the same object. If such information is not available, worst-case assumptions have to be made in order to create correct code. In the C language, aliasing can occur by use of pointers. As pointers are generally a very frequently used feature of C and also array accesses are just disguised pointer arithmetic, alias analysis is very important. An early discussion can be found in [149], more recent research in [96], [43] and [94].

C compilers usually can use the following methods to obtain aliasing information:

- The C language does not allow accessing an object using an lvalue of a different type. Exceptions are accessing an object using a qualified version of the same type and accessing an object using a character type. In the following example `p1` and `p2` must not point to the same object:

  ```
  f(int *p1, long *p2)
  {
    ...
  }
  ```

  A C compiler may assume that the source is correct and does not break this requirement of the C language [79]. Unfortunately, this is certainly a lesser known rule of the C programming language and may be broken frequently by programmers.

- Further information on possible aliasing can be obtained by data-flow analysis. A typical way to obtain that information is to solve the "points-to" problem by iterative data-flow analysis. Rules imposed by the programming language are used to model the effects of an instruction on possible aliasing.

  There are slightly different ways to calculate that information as well as different scopes for application of this analysis, e.g. intra-procedural or inter-procedural,

flow-sensitive or flow-insensitive. In the following example, alias analysis will detect that `p1` can only point to `x` or `y` whereas `p2` can only point to `z`. Therefore, it is known that `p1` and `p2` are not aliased.

```
int x[10], y[10], z[10];

int f(int a, int b, int c)
{
  int *p1, *p2;

  if(a < b){
    p1 = &x[a];
    *p1 = 0;
  }else{
    p1 = &y[b];
  }
  p2 = &z[c];

  ...
}
```

Flow-sensitive analysis would also reveal that the array `x` may be modified by the code shown above, whereas the array `y` is not modified. Flow-insensitive analysis does not deliver the information that `p1` can only point to `x` at the location of the assignment.

This analysis produces less problems with common broken code, although seriously "ill-minded" constructs like using pointer arithmetic to step from one array into another are likely to confuse such analysis.

It should be noted that due to the complexity of the "points-to" sets, the data-flow analysis for context-sensitive alias analysis is one of the more expensive (in terms of memory and time) data-flow problems to solve.

- The 1999 C standard provides the `restrict`-qualifier to help alias analysis. If a pointer is declared with this qualifier, the compiler may assume that the object pointed to by this pointer is only aliased by pointers which are derived from this pointer. For a formal definition of the rules for `restrict` see [79].

  A very useful application for `restrict` are function parameters. Consider the following example:

```
void cross_prod(float *restrict res,
                float *restrict x,
                float *restrict y)
{
  res[0] = x[1] * y[2] - x[2] * y[1];
  res[1] = x[2] * y[0] - x[0] * y[2];
  res[2] = x[0] * y[1] - x[1] * y[0];
}
```

Without `restrict`, a compiler has to assume that writing the results through `res` can modify the object pointed to by `x` and `y`. Therefore, the compiler has to reload all the values on the right side twice. Using the hints given by the `restrict` keyword allows the following transformation:

```
void cross_prod(float *restrict res,
                float *restrict x,
                float *restrict y)
{
  float x0 = x[0], x1 = x[1], x2 = x[2];
  float y0 = y[0], y1 = x[1], y2 = y[2];

  res[0] = x1 * y2 - x2 * y1;
  res[1] = x2 * y0 - x0 * y2;
  res[2] = x0 * y1 - x1 * y0;
}
```

So far, use of the `restrict` keyword in existing code is rare, not many programmers know of that possibility and only some compilers correctly implement the `restrict` qualifier. The main rationale for adding it to the C language was the necessity to make good code generation for numerical code possible (the aliasing rules of C forced some serious disadvantages compared to Fortran, see [80]) — the impact on typical code for small embedded systems surely will be much smaller. Nevertheless it is an additional source for obtaining more precise aliasing information and should not be dismissed easily.

It is important to note that all the three sources for obtaining aliasing information usually give additional information and no single source can subsume the others.

As alias analysis is not a transformation per se but only an analysis that helps when applying many optimizations, it can not have a negative effect as such — additional information is never harmful in itself. As alias analysis often enables more traditional optimizations, it is a promising analysis for small embedded systems (and other systems as well) if efficient code is to be generated. vbcc performs context sensitive intra-procedural alias analysis, supported by some inter-procedural data-flow analysis.

3.3.13 Inter-Procedural Data-Flow Analysis

Many of the optimizations mentioned so far require data-flow information to be applicable on a larger scope (i.e. more than basic blocks). Several of them (e.g. the loop optimizations) always require such information to be applied to any but the most simple cases. Data flow information can be obtained, for example, through classical iterative algorithms or via SSA form.

Typically, data-flow problems are solved for each procedure separately. Therefore, worst-case assumptions are used for every procedure call. For example, in C it is usually necessary to assume that all global and static variables as well as all dynamically allocated memory and all local variables whose addresses have been taken may be accessed in a function call.

Therefore, it is beneficial to perform some analysis across procedures. This can be done by modelling the effects of a function call on the data-flow information that is computed in a similar way as it is done for every statement or basic block. This modelling can be done in various granularities. In the simplest case, the sets of variables considered to be read and/or modified by a procedure call are reduced. Similarly, tracking the registers used by a procedure enables better register allocation as values can be stored in caller-save registers across a procedure call if the callee does not actually use these registers.

More detailed results can be obtained if data-flow information created by the procedure is also taken into account. For example, after a call to a function that always sets a variable to zero, constant propagation may propagate this value into the calling function. Conceptually, this can be done by calculating corresponding "gen"- and "kill"-sets for the function as is done for every basic block in conventional data-flow analysis.

Further detailed information can be obtained if the analysis is done context-sensitive, i.e. the "gen"- and "kill"-sets are recalculated at every call site taking into account the actual arguments for this procedure-call.

Inter-procedural data-flow analysis is presented in [6] or [16]. Recent research can be found, for example, in [56]. Application of inter-procedural analysis is examined in [129]. Inter-procedural constant-propagation is presented e.g. in [28] or [29]. Register allocation across procedures is described, for example, in [146] or [58].

Unfortunately, inter-procedural data-flow analysis can have rather big time-complexity as the data-flow information for all callers may have to be recomputed every time more precise information was calculated for a procedure. It only gets worse if further analysis, e.g. alias analysis, is used which may, in turn, be calculated inter-procedurally (see [149, 43, 94]). As shown in [107], the complexity of inter-procedural data-flow analysis in the presence of recursions, pointers and aliases can be exponential. Of course, it is possible to omit all this recalculation and obtain less precise information.

A special case are non-recursive programs. Such programs, which are typical for the type of embedded systems that are aimed for in this thesis, have an acyclic call-graph. By visiting all procedures in topological order (starting with the leaf functions), context-insensitive inter-procedural data-flow analysis can be computed in a single pass without too much additional overhead. vbcc uses this approach and computes sets of variables and memory locations that can be read and written by functions. This information is then used to obtain more precise data-flow information when optimizing the call-sites.

Therefore, for small embedded systems, the overhead of inter-procedural data-flow analysis seems reasonable. Especially, as inlining usually will be used less often. Compilers for bigger systems tend to inline many of the smaller leaf functions, therefore implicitly calculating "inter-procedural" data-flow information for these inlined calls.

These leaf functions are typically the most promising candidates for inter-procedural data-flow analysis. Complex non-leaf functions often contain too many side-effects that can not be analyzed and do not benefit much. So, if the small leaf functions are not inlined, e.g. due to code size constraints, inter-procedural data-flow analysis can help significantly to generate better code for the call sites.

3.3.14 Cross-Module Optimizations

Traditionally, compilers for many programming languages (and specifically C) translate several source files or modules separately, producing a relocatable object file for every input file. This separate compilation makes faster turn-around times possible. After modifying one module, only this module has to be recompiled in many cases.

A separate program, the linker, produces an absolute executable (or ROM image in our case) from these modules. As it performs only simple tasks of calculating addresses of symbols and filling them in according to usually very simple rules, linking is very fast — much faster than compiling or optimizing.

Apart from faster turn-around times, memory consumption is also reduced as the compiler only needs memory to translate the modules, but not for the entire program. These advantages were very important when the machines that did the compiling were less powerful than today. Also, for huge applications running on personal computers, workstations or servers, these factors are relevant. For the small embedded systems we are targeting, however, these issues are less important today. The programs are not as large because the targets have only low memory capacity, whereas the host machines used for cross-compiling the code are several orders of magnitude more powerful these days. Therefore, it is interesting to examine the trade-offs one gets by using separate compilation.

- To perform function inlining as described above, the compiler needs to see the code for the function to be inlined. Compilers looking at each source file separately, obviously are very limited in this optimization. The ability to inline files from other modules can be very beneficial.

- Inter-procedural data-flow analysis also requires the compiler to see the code of the procedures called. Only if the callee can be analyzed (including its callees), precise information can be used at the call site.

  Therefore, similarly to cross-module function inlining, the more procedures the compiler sees, the more use it can make of inter-procedural data-flow analysis.

- A further possibility to reduce size of the output is to remove objects that are not used anywhere in the code. This includes, for example, variables that have been forgotten or are unused because only certain functions of a library or module are used. Also, procedures that have been inlined in every call, can be eliminated.

  In simple form, this has been a feature of most linkers for a long time. Object modules in a library are not included in the final executable unless they are referenced. Most linkers, however, were not able to leave out parts of a module. Some more modern linkers are able to do this, but it requires some support from the compiler as well certain features of the object file format (e.g. every object has to be put in something like an own section).

  A compiler seeing the entire code of the application, however, is able to eliminate all unused objects, even if the object file format does not offer further support.

  Contrary to the other cross-module optimizations mentioned, however, in this case it is really necessary to see the *entire* code. While the other optimizations benefit gradually as more source code is available, the view of the entire code basically is necessary for this optimization to work at all.

  Also, especially for embedded systems, there often will be objects that are not referenced by the source code but are nevertheless needed. For example interrupt

handlers or version information that is supposed to be in the ROM. These objects will have to be attributed somehow to prevent the compiler from eliminating them.

So, altogether this optimization can be useful but has some problems in practice. If it is possible to implement it, it can be helpful though. A clever linker may be able to achieve a similar effect more easily, but also a little bit less efficient (see, for example, the next item).

- A further limitation of separate compilation that is less obvious, is the negative impact on machine code generation. Producing one object file for many files rather than many separate objects, sometimes enables the compiler to select smaller and/or faster instructions.

  Assume, for example, a machine that has two instructions performing a subroutine call. Let one instruction be a relative branch that takes up two bytes of space and can call a subroutine which is less than 16KB distant from the location of the branch. The other instruction could be an absolute jump that is four bytes large and can reach any subroutine from anywhere. This is not an unusual assumption for real architectures.

  If the compiler now has to generate a call to a procedure in a separate module, it has to generate the larger absolute jump, because it does not know whether the target will end up close enough in the final executable. Generating an object containing the code for the callee as well as the call-site, enables the compiler (or assembler) to check whether each call is located close enough to its target, and to generate the smaller branch if possible.

  Similar situations exist for other instruction patterns, obviously depending on the target architecture.

- Furthermore, additional to exploiting optimization possibilities as mentioned in the item above, the opportunities can also be increased by smart placing of code and objects.

  For example, assuming the situation described above, placing code for called procedures close to the call sites may exhibit more opportunities for using the smaller branch instructions.

  Also, placing static variables that are often used together close to each other, may allow generation of better code on some architectures.

- Another lesser known optimization that is well suited to small systems is the smart placing of static data to reduce memory overhead due to alignment requirements.

  For example, many compilers will place static data in memory in the order of appearance in the source code. As most architectures require data of certain type to be aligned, sometimes they leave space unused between objects with different alignment requirements.

  Consider a machine with a pointer-size of four bytes and natural alignment requirements (i.e. every type has to be aligned according to a multiple of its size). Let us look at the following code:

```
char *input_ptr, input_idx;
char *output_ptr, output_idx;
char *copyright_string;
```

Placing the variables as they appear, will yield a memory need of 20 bytes. As the single-byte characters are followed by objects with an alignment requirement of four bytes, three padding bytes have to be filled in after each character.

If the pointers are placed first, followed by the single-byte characters, only 14 bytes of memory are required. Therefore, this is an easy and promising optimization to save space. It can also be done as an intra-module optimization, but will be more efficient when done across modules. This is especially true if there are many different sections for different kind of variables (e.g. the PowerPC Embedded ABI [122] specifies `.text`, `.data`, `.bss`, `.rodata`, `.sdata`, `.sdata2`, `.sbss` and `.sbss2` just for normal data, not considering specially attributed data like vector tables etc.).

This is a list of more promising cross-module optimizations that are recommended for small embedded systems. Although such optimizations are relatively new in production compilers, many of them are offering at least some cross-module optimizations now, e.g. [70], [76] or [128]. Recent research is presented, for example, in [12], [21] or [66]. See [41] for a discussion of the benefits of cross-module optimizations for embedded systems.

Also, research and some products are available regarding tools that do optimizations after link-time, i.e. on the final binary image (see, for example, [146], [63] or [2]). Such an approach may offer some additional possibilities for optimization, but it lacks the high-level view the compiler has. Another problem is the reliable analysis of machine code. Usually, this is only possible if there is some knowledge about the compiler that generated the code (see also section 4.2.3). Anyway, it underlines that there is something to gain with cross-module optimizations.

The two major problems with these optimizations are probably the transparent handling to the user and the unavailability of source code. C programmers are used to the separate compilation and linking steps. They often use complicated makefiles and build processes, parts of the application may only be available as object code etc. To get accepted, cross-module optimizations should require no changes to this infrastructure. It should be possible to switch them on easily, for example by a simple compiler option as described in section 2.3.4.

This leads to the other problem, unavailability of source code. While most optimizations mentioned above also work if only part of the source code is available, their impact grows with the amount of source code that is available. For example, if a big part of the functionality is covered in library functions that are only available as object code, the effect is limited.

Luckily, especially for small embedded systems, this is rarely the case (see section 2.1.8). Typical code for such systems is largely written by the manufacturer of the device. Even the parts provided by other companies (for example communication drivers) are mostly delivered in source code. Due to the stringent memory constraints, object code often does not offer enough scalability to exactly fit the need of different users.

Source code with conditional compilation features is a prominent solution here. Also, liability issues and frequent changes of target architectures (compared to personal computers or servers) favour the use of source code.

To sum it up, cross-module optimizations can increase the efficiency of the generated code and are pretty well suited, especially to small embedded systems.

3.4 Combination of Results

To conclude the results from the last sections, all optimizations that can increase RAM usage were turned off. This includes:

- common subexpression elimination
- loop invariant code motion
- strength reduction
- register allocation of constants
- unrolling of loops with a non-constant number of iterations

All Optimizations of vbcc that do not in themselves increase RAM usage have been performed, especially (but not limited to) the following optimizations that often also reduce RAM usage:

- dead assignment elimination
- function inlining
- loop unrolling
- global constant propagation
- global copy propagation
- inter-procedural register allocation

Table 3.11 shows the RAM usage that was obtained using this combination of settings intended to reduce RAM usage. It was indeed possible to reduce total RAM usage of the test cases by almost 20%, just by choosing a certain selection of common optimizations and parameters. However, this gain was traded for execution speed and code size. Also note that even with these settings, there is still one test case where RAM usage has increased over the default optimization settings. Also note that this rather unusual combination of optimizations is unlikely to be found in current compilers that either optimize for speed or for code size. For example, a compiler optimizing for speed will surely employ aggressive common subexpression elimination whereas a compiler optimizing for code size will never perform aggressive function inlining. Therefore, a new option “optimize for RAM usage” is proposed.

Table 3.11: Combined Results

Func	**Ref.**	**Combined**	**%**
s241	560	480	86
s243	320	272	85
s626	144	48	33
s757	160	128	80
s81	304	240	79
s84	288	176	61
s86	32	48	150
total	**2272**	**1856**	**82**

3.5 Conclusion

The discussion of common optimizations has revealed several key points:

- RAM (stack) usage can be affected by common compiler optimizations. While this is not significant for large systems, it can be of interest for very small systems with tight memory constraints.
- The three goals of execution speed, code size, and RAM usage are often mutually exclusive.
- The combination of optimizations that resulted in least RAM consumption is different from those that would be chosen when optimizing for speed or code size.

These tests show that optimizing for RAM usage could be a useful additional option in compilers for small systems. Some first directions have been shown, but clearly this only scratches the surface. There are more sophisticated versions of the optimizations presented, and their impact can vary with the way in which they are implemented. As the optimizations performed by vbcc are reasonably common and largely comparable to those done by many other compilers, similar results are expected with most optimizing compilers. Nevertheless, it would be interesting to conduct these tests for other compilers. Different combinations of options could also be examined, maybe on a per-function basis. Approaches like those described in [10], [44] or [88] could be used to automatically find combinations of transformations that deliver the best results with respect to RAM usage.

Chapter 4

Stack Analysis

One important topic, which is both a safety and efficiency issue, is determining the amount of stack space to use for an embedded system. Code size and the size of static data can be easily calculated (e.g. by the linker) and, in fact, have to be known to produce a working binary. While dynamic (i.e. heap-allocated) memory is very hard to calculate, it is not used in small systems using static operating systems (see section 2.1.8).

Stack space, however, is used by almost every system and there are hardly any C compilers that can generate code not using a stack. Although there are no recursions in the systems considered here (see section 2.1.8), a stack is still necessary to have reentrant code. For a multi-threaded system, using preemptive tasks and interrupts, reentrancy is crucial. In fact, non-reentrant library routines delivered with compilers often cause problems and have to be replaced.

On larger systems, programmers rarely have to think about the stack usage of their applications. Usually, some of the following reasons allow them to ignore that problem:

- Most big data-structures are allocated on the heap. Basically, large arrays are the only large data-structures that can be allocated on the stack in C (structures are not that big unless they contain arrays). Complex (i.e. connected with references) or dynamic data-structures can only be allocated on the heap.

  So, even a few hundred KBs of stack-space is sufficient for most programs. This amount of memory can easily be reserved as stack space by default on modern personal computers or workstations. Even recursive programs rarely hit that border.

- Many operating systems rely on a MMU, hence they can detect a stack overflow and are able to extend the stack easily by mapping another page of memory. In such systems only free memory and address-space are a limiting factor.

- On systems without a MMU, a compiler, can still provide automatic stack extension. It just has to insert additional code whenever a new stack-frame is needed (typically at a function entry). This code has to check whether there is still enough stack space and allocate more space. In that case, the stack does not have to be contiguous.

On small embedded systems none of these solutions are possible. The first one obviously gets ruled out due to the very tight memory constraints. Small systems also do not have a MMU — even if they had one, the memory loss caused by page sizes

and alignment can not be tolerated on systems with less than 32KB of RAM (smaller MMU pages would cause larger overhead for the MMU tables).

The stack-extension by compiler-checks can also not be used as it requires the availability of free RAM (and adds some runtime overhead which, however, would be more tolerable in most cases). Having unused spare RAM on such a system increases costs. As RAM makes up a big part of the total chip cost, it is essential to use as little RAM as possible. Therefore, a close upper bound of the stack space needed is most desirable.

Too small stack areas, however, can be even more costly than wasted space. Stack overflows can not only crash a system but they can make the system behave erroneously and unpredictably in very subtle ways. Instead of a complete crash that may trigger a watchdog that reboots the systems, stack overflows might, for example, cause an engine to stutter or, one of the worst cases, an airbag to explode.

Even smaller failures in less important systems may cause an error lamp to be switched on and require that the car be serviced. If such issues require the callback of an entire series, enormous costs will be incurred by perhaps a few bytes of missing stack space (see [86] for recent statistics on callbacks of cars).

As a result, stack analysis yielding safe upper bounds for stack usage of application tasks is not only needed to avoid costly failures due to stack overflows, but also to make better use of available RAM.

In this chapter, after presenting the existing practice, a good solution for this problem will be shown. A relatively simple extension to the compiler is made, making use of sophisticated static analysis already used for the optimizing techniques discussed in the preceding chapters. It will be shown that it is easy to implement (if the required infrastructure for global cross-module analysis is available), relatively easy to use for the application programmer, and delivers good and reliable results.

4.1 Existing practice and Related Work

When using a static operating system, the application programmer at least has to think about stack sizes as he will be prompted for it during/prior the generation phase. Either he will have to specify a single stack size or separate sizes for each task or interrupt service routine.

At this point, many developers are rather clueless as they have no idea how much to specify. Actually, even a rough guess requires rather intimate knowledge of the processor architecture in use, the ABI, and the compiler generating the code. A two-fold or more increase in stack usage for the same source code is not uncommon when switching from one architecture (e.g. a small 16bit microprocessor) to another one (e.g. a 32bit microprocessor).

Therefore, the problem arises every day and has to be addressed in some way. This section will present some typical approaches that are used in practice and discuss what tools are available to help with those issues.

4.1.1 Guessing and Testing

One common approach which is unfortunately still frequently used is simply to guess the amount of required stack space. A value which seems plausible to the developer is used and the system is tested. If it seems to work, the stack size is kept that way or reduced and the test is rerun. There are several serious problems with this approach:

- The first problem here is the "educated guess" of the developer. Frankly, few programmers or computer scientists, can give a decent estimate on anything but the most simple programs by looking at the source code. They usually base their guess on prior experience on other architectures or different compilers. This may work a few times but may fail seriously as mentioned above.

  Also, as soon as the code is written by different people or even companies, or if code from external libraries was included, it will take quite some amount of work to even make an educated guess.

- The next problem is the reliability of tests. Usually, it is assumed that the stack space seems to be big enough if the system passes tests. However, is it really known that the worst-case of stack consumption was actually encountered during the tests?

  Basically, determining the situation with the highest stack usage is almost as hard as calculating the stack size. It requires detailed analysis and is rarely done. Instead, it is assumed that running long tests will encounter the worst-case with a high probability.

  However, especially in multi-threaded and interrupt-driven systems, the worst-case situation may occur only under very special conditions, for example one task may have to interrupt another task at the point of its highest stack usage and must itself been interrupted by one specific interrupt when it has reached its own worst-case.

  If we keep in mind that often a word is pushed on the stack only for one or two instructions (e.g. if a register has to be spilled) and the systems in question run with clock-frequencies from several MHz up to 100MHz, the period of worst-case stack consumption of a single task may be shorter than a microsecond. On the other hand, the period of such tasks or interrupts may be several milliseconds for cyclic tasks and even longer for special event-triggered interrupts.

  As a result, encountering the worst-case can take arbitrary long times of testing. Obviously, if the worst-case is so rare that it occurs neither during testing nor during the life-cycle of the product, there is no real problem. However, in situations like the automotive industry, the operating hours of a system can be much larger than the hours of testing. Consider, for example, a small to medium car manufacturer producing about one million cars per year. If one ECU is used in half of this manufacturer's products during six years of production, there are three million of them on the road. With an average of 3000 operating hours per car all exemplars in total may run 10^{10} hours — several magnitudes over what can possibly be achieved during testing.

- Furthermore, as it was already mentioned above, stack overflows can cause very subtle misbehaviour and are not always recognized as such. Therefore, a lot of development time may be wasted tracing program bugs that turn out to be simply stack overflows. Especially when operating systems are used, the application programmer may be unable to understand the effects of the overflow, and some system variables that may have been overwritten just cause strange behaviour of the operating system. The author has been involved in several such cases that turned out to be stack overflows.

  Also, the stack overflow may not cause any immediately visible harm and therefore it may go unnoticed during the testing phase. As a result, the developer thinks

the stack was sufficient although there already was a stack overflow.

- Still, if the stack size was correctly guessed and if it is sufficient even in the worst case, it may be significantly too high. It is very unlikely that a stack size determined by those means is close to the real upper bound. Either it will be too small for the worst case or it will waste quite a bit of valuable RAM.

  Reliably guessing the stack size needed by a non-trivial task with a precision of more than about +/− 16 bytes seems unrealistic even on easier architectures. So it seems natural to assume that someone who always manages a safe upper bound that way is likely to waste 0 to 32 bytes per task stack. With 10 to 20 tasks, it is not uncommon to have 1/10th of the entire RAM (maybe 2–4KB) of the system wasted just by unnecessary big stacks.

- Finally, this process of guessing and testing is also time-consuming. Changes to the software or build environment will make it necessary to re-do most of it. New functions added, changes of timing, different compiler options or new versions of libraries all can significantly change the stack usage.

4.1.2 High-Water Marks

An improvement to the approach described above is the introduction of "high-water marks". At the startup of the system, all stacks will be filled with a certain bit-pattern (e.g. 0xAA). Then tests will be run just as in the previous method. After those tests, the stacks are examined and by determining how many bytes in the stack still contain this bit-pattern, the maximum stack usage during this test-run can be calculated for each task.

This approach has several advantages:

- The trial-and-error phase of the previous approach (reducing/increasing the stack-size and re-running tests) to minimize stack usage is somewhat accelerated. While the last approach only gives a qualitative result (stack seemed to be sufficient or not), the high-water mark yields a quantitative result (how many bytes were unused). Therefore, one gets a better starting point for trying other stack sizes.

- A stack overflow is easier to detect as all the bytes have a fixed initial value. If the end of the stack was overwritten, a stack overflow is very likely. This may reduce the danger of unnoticed stack overflows during testing.

However, there are still major problems with this approach:

- If the application causes a stack overflow, but does not write all bytes of the stack, the overflow may go unnoticed. Consider, for example, the following code:

```
void f(int n)
{
  int a[10], i;
  for(i = 0; i<n; i++)
    a[i] = ...
  ...
}
```

If the stack was already nearly full, the function will further move the stack pointer to create space for the array. However, if `n` was small it would perhaps only write to the bytes outside of the stack, but would leave the actual end of the stack untouched (assuming the stack grows downwards — otherwise a loop that counts down would cause this behaviour).

- The problem of encountering the worst case during testing is still there, just as in the previous approach. No improvements are made in this area.

- Similarly, the problem of wasting space is not much improved. One only obtains the stack usage during the test runs. So there is still the question of either using more or less this value (with the possible risk of failure during the product lifetime) or adding some amount of "safety-buffer".

- The problem of maintainability is also not improved. After almost any modification to the software, tool-chain etc., the tests have to be re-run in a way that suggests at least some reasonable chance of encountering the worst-case.

4.1.3 Manual Analysis

Sometimes (usually when systems are safety-critical or RAM space seems to be too small) a thorough manual analysis of stack usage is conducted. The machine code (or assembly language) generated by the compiler is analyzed as well as the source code. With a lot of work (and perhaps the limited help of some tools) a call-tree can be constructed. Then, by extracting the stack usage of separate functions from the generated code, a stack analysis of entire tasks is possible.

Note however, that this analysis basically has to be performed on the assembly language or maybe even machine code level. The source code can only be used as a reference or hint. Modern compilers perform too many transformations that destroy any straightforward mapping from source code to object code (as has been shown in chapter 3). Inlining of function-calls, introduction of new temporary variables and countless other transformation will have significant impact on stack size.

Nevertheless, this is the first approach able to actually yield precise and reliable information. However, there are many grave problems:

- First of all, there are few people who are able to execute such an analysis reliably. It requires not only a good understanding of the application in question, but also in-depth knowledge of the processor architecture and the code generation of the compiler.

- Even then, it may be extremely hard to understand the code generated by a sophisticated optimizing compiler. For example, the order of functions in the machine code may be completely different from their order in the source code. Functions may be eliminated or different specialized versions of one function may exist. Add to that, the functions and variables without external linkage. Usually, they will just have numbered labels and it is obviously extremely hard to find the code for a static function in a large piece of optimized machine code.

- Probably the most important drawbacks are the amount of time and work that is needed to carry out such an analysis, together with the high risk of mistakes. Humans are still much more prone of errors than software. Optimizing compilers are a good example here. Although they are amongst the most complex pieces

of software ever written [106] and bugs in them are, in fact, found every now and then, even the best programmers will make several orders of magnitude more mistakes than they will encounter compiler bugs.

- Because such manual analysis is so much work, the fact that it pretty much has to be redone every time there are more than trivial changes to the software, weighs even more. The same is true for change of compiler options, compiler version or even when switching to another compiler or target architecture.

As a result, real manual analysis (obviously most people performing the before mentioned "guessing and testing" will tell you they were doing something like manual analysis) is very rarely done and very expensive. Still for small and very critical projects it may be the method of choice — especially if you do not want to trust any tools.

4.1.4 Traditional Compilers

Traditionally, C compilers have at least some knowledge about the stack usage of the programs compiled by them, especially on a per-function basis. Standard C conforming to the ISO 9899:1989 standard (equivalent to "ANSI C") does not allow data of dynamic size with automatic storage duration (i.e. on the stack). The new revision of this standard ISO 9899:1999 [79] does support variably sized arrays with automatic storage duration, however this standard is still not fully supported by most compilers and dynamic arrays are not used in the systems talked about here.

Therefore, a C compiler can (and will) translate every function in a way that it can only use a maximum constant number of bytes on the stack and will pop them all off before the function is left again. Some ABIs even prescribe that the entire stack space of the function will be reserved (or pushed) at the prologue of the function and no further stack-pointer manipulations are made until the epilogue (see [122]). When targeting such an ABI, a compiler must in fact be able to calculate the stack usage of a function to be able to generate correct code at all.

Also, many compilers provide features like dynamic stack-checking or stack-extension. To implement those features, the stack usage of a function must also be calculated.

So, while it is obvious that most C compilers are able to calculate this information, there remain some questions. First, do they provide this information to the user and, second, can they calculate the stack usage for an entire task?

Regarding the first question, many — but far from all — compilers do provide this information in one way or the other (even if it may not always be easy to get it). It may be written into a report file, output as a linker symbol, extracted from the assembly code or similar.

Unfortunately most tasks — especially the complex ones where calculation of stack usage is difficult — are not made up of single functions. They will call subfunctions, code from other vendors, inline assembly, maybe function tables, etc. The information is mostly useless when only available for single functions rather than the entire call tree.

Almost all of today's production C compilers — especially for embedded systems — still translate programs on a file by file basis (actually, for the most part even on a function by function basis). As a result, they are unable to build a call-tree for any but the smallest programs (and therefore do not even try in most cases).

There are a few simple attempts to output the stack-usage for each function during compilation and then compile them to a complete call-tree with stack-usage information (e.g. [46, 125, 47]). A simple approach is to generate the stack usage of functions as

linker symbols and let the linker build a call-tree and calculate the stack usage for entire tasks.

However, there are many shortcomings to these approaches. The stack usage of single functions and referenced linker symbols is simply not enough information for this purpose. Even the simplest cases of calls through function pointers will confuse such an implementation. As a result, this approach does seem much too restricted and unreliable.

4.1.5 Post Link-Time Analyzers

To overcome the problem traditional compilers are having due to their separate compilation, it was proposed to perform this analysis after linking, i.e. when a final binary image has been produced from all modules. The idea is to read the final executable (or, very similar, assembly code as described in [82]), reconstruct a call-tree and flow-graph for the entire system and then compute the stack usage, for example using abstract interpretation (see [124]). For a critical comment on abstract interpretation, see [32]. Stack usage analysis to improve dynamic stack allocation on large systems based on machine code is presented in [66]. Commercial tools can be found in [2] and [22].

This mechanism can have several advantages:

- The entire code, including assembly parts, libraries only available as object code, etc. can be analyzed by the tool.
- In theory, stack analysis can be done to code generated by any compiler even if the compiler does not do any stack analysis.

There are, however, serious problems when dealing with machine code and performing static analysis. First, there are many issues that can significantly complicate constructing the control-flow and call-tree of such code. A few such examples are:

- Use of function pointers. Even though heavy use of function pointers (e.g. function pointer variables that are passed as function arguments) should not be used in the systems that are examined here, some simpler cases are often used. One example is calling a function from a constant array of function pointers to implement a state-machine.

```
int (*const (state_fcts[]))() = {f1, f2, f3, f4, ... };

void state_machine()
{
  int state = 0;
  while(1)
    state = state_fcts[state]();
}
```

  Without high-level information, it is very hard to determine that the possible targets of such a call are the ones pointed to in the array `state_fcts`.

  It gets even more complicated if the array contains not only function pointers:

```
const struct {int state; void (*fp)(); } fa[] =
                        {1, f1, 2, f2, 3, f3, ... };
```

Either very detailed analysis of the code accessing the array would be necessary or all entries in the array that might point to executable code have to be considered as call targets. However, in many cases this can not be detected. For example, many embedded systems have code in memory starting from address 0 onward — an integer like 20 would have to be considered as a possible call-target, potentially confusing analysis beyond any repair.

Even if an access to a function-pointer array can be detected, it will usually be impossible to determine the beginning and end of the array in many cases. After all, there can be arbitrary code or data located around the array. The compiler might not even use the real address of the array but an already offset one, e.g.

```
int (*const (state_fcts[]))() = {f1, f2, f3, f4, ... };

void state_machine()
{
  int state = 0;
  while(1)
    state = state_fcts[state + 1]();
}
```

Here, the compiler could use the address `state_fcts + 1` (which is a constant) as base address, saving the addition of 1 to `state`. Therefore, the address appearing in the machine code is not the start of the array but rather a pointer to somewhere inside it.

- Another tricky problem when dealing with machine code is branches into the middle of one machine instruction. For example, on the Motorola 68k series of processors and microcontrollers the following more or less common optimization is used by some compilers to replace a branch instruction and get better pipeline usage. For example, a C statement like

```
if(a > 5)
  res = 12;
else
  res = 21;
```

could be translated to these 68k instructions:

```
        cmp.l   #5,d1
        bge     l1
        moveq   #12,d0
        bra     l2
l1:
        moveq   #21,d0
l2:
```

Now, the `bra l2` can be replaced by, for example, a compare instruction with an immediate operand that is the opcode of the `moveq #21,d0` instruction. Actually, this opcode is 0x7015 yielding the following optimized code:

```
        cmp.l   #5,d1
        bge     l1+2
        moveq   #12,d0
l1:
        cmp     #0x7015,d0
l2:
```

Note the modification of the first branch into the middle of the `cmp #0x7015,d0` instruction and the removal of the unconditional branch `bra l2`. Actually, the length of the code is unchanged. Only the unconditional branch has been replaced by the first half of the new `cmp`-instruction.

When the first branch is taken, execution continues at the same address as in the first version which still contains the opcode for `moveq #21,d0` — only hidden inside the compare instruction. If the branch is not taken, after executing the first `moveq`, the new `cmp`-instruction will be executed. Assuming that the condition codes are not evaluated afterwards (which is a precondition to perform this transformation), this instruction does not have any noticeable effect. The second `moveq` is basically turned into a `nop` in this path.

Similar transformations are possible on many architectures with variable instruction lengths and there are, in fact, compilers actually producing such code. Constructing the control-flow for code like this, however, can be a problem (although there are ways to handle it).

- The code generated for `switch-case` statements can also impose difficulties when analyzing machine code. There are many different ways to generate code for such statements.

  In the simplest case, a series of compare- and branch-instructions is emitted (one for every `case`). This will not be a problem. If, however, there are many `case`s, many compilers will resort to other code patterns.

  Series of `case`-labels with continuous values are often translated into jump-tables. The value controlling the `switch` will be transformed into an index that is used to fetch the corresponding entry from an array of jump targets. These targets can be stored in some section for constant data, but are also often contained directly after the code generated for the `switch` statement. Both, absolute addresses as well as (PC)-relative addresses might be used.

  Basically, the problem of analyzing this is similar to the problem of calling functions through an array of function pointers. Theoretically, it is somewhat better to analyze as the generated code usually has to do an explicit bounds check before going through the jump table which could be used to determine the array boundaries. In practice, however, this is quite hard in the general case. More realistically, pattern matching could be used to recognize code usually emitted for such statements. A few remarks on pattern matching will be made below.

  Many compilers also use library routines to handle `switch` statements. The compiler will emit a constant data structure consisting of the jump targets and the values corresponding to them. The address of this data structure is then passed to a library function together with the value controlling the `switch` statement. This library function will, for example, do a linear or binary search through this table and indirectly jump to the right target.

Again, correctly analyzing such code seems almost only possible by recognizing this code and putting knowledge about it into the analyzer.

- Some smaller microcontrollers use a segmented code and/or data space, e.g. a 16bit processor supporting a 24bit code space using segment registers. Addresses may not fit into a single register or instruction, complicating work of the analyzer. Furthermore, calling a function or returning from a function may require a sequence of instructions or even calls to library functions (for example the Tasking compiler for the C16X uses library calls for indirect huge-pointer calls). Again, analysis requires knowledge of this specific compiler's code generation patterns and library functions.

Apart from these problems constructing the control-flow, there are some general problems:

- As has been shown above, many common constructs will require some kind of pattern matching to detect them. If these patterns are changed, e.g. because different compiler options or preconditions are set, they are likely to be missed. Therefore the effectiveness of such pattern-matching heavily relies on intimate knowledge of code-patterns generated by specific compilers. Usually only non-exhaustive tests will be possible if the vendor of the stack analysis tool is not the compiler vendor. Therefore, it is likely that the pattern-matching may sometimes fail.

- Similarly, compilers that perform instruction scheduling may further complicate pattern-matching. As soon as the piece of code to match is not fixed but may vary slightly, it gets difficult to match all the right pieces without introducing some "false positives". There is a great danger of finding pieces of code that are very similar to the code patterns that are searched for, but nevertheless have different semantics. Such a case may lead to silent incorrect results which is fatal.

  Certain incorrect optimizations of parts of the SPECCPU [131] benchmark suite provide historical evidence of the problems of pattern matching code. Apparently compilers emitted manually optimized code for certain small functions that have high impact on the benchmark results. Tests revealed that, among other problems, this pattern-matching sometimes also matched similar but different pieces of code and therefore resulted in corrupt code. For further details, see e.g. [109], [132], and section 4.2.3.

- There will often be cases that can not be analyzed completely automatically, e.g. certain calls through function-pointers. There will have to be annotations made by the programmer. However, a tool working on machine code has no high-level information left. It will usually not be able to tell the developer the line of (source) code that causes the problem but rather a memory address of a machine instruction.

  An application developer has to have pretty deep insight into the processor architecture used, the way the compiler generates code, and the entire application. Consider, for example, that an indirect call causing problems may have been inside a function that got inlined into another one or it may have been inside a loop that got unrolled.

  Consider even simple code like:

```
void (*const (fa1[]))()={f1, f2};
void (*const (fa2[]))()={f3, f4};

void test(int i)
{
  if(i<2)
    fa1[i]();
  else
    fa2[i-2]();
}
```

Probably an analyzer tool will not be able to calculate the possible call targets itself and report two indirect calls it can not resolve. As compilers often move around code in situations like this, it is not clear whether the first call appearing in the machine code corresponds to the first one appearing in the source code. To correctly annotate these calls, the user has taken a very close look at the generated machine code to find out which of the two calls has the possible targets `f1` and `f2` and which one may call `f3` or `f4`.

While a programmer proficient in machine language may be able to handle such a simple example, understanding larger pieces of heavily optimized compiler generated code can be extremely difficult.

Also, in the absence of function identifiers, the situation gets very tedious. While, depending on the file format of the produced binary, exported labels may be visible, identifiers of static functions are usually not contained in the final binary anymore. Reconstructing the assignment between static functions and pieces of machine code will not be easy.

If a switch statement compiled into an indirect branch is not detected by a binary analyzer, extracting the possible jump targets from the machine code and creating annotations certainly seems way above what can be expected by a normal application developer.

- Similarly, there is the problem of specifying these annotations. First, they usually can not be put inside the source code (which would probably be the best place) or the final binary. Therefore, extra files containing the annotations will be required. However, there is no more connection to the source code.

  So, even if it is known, for example, that the code contains two indirect calls and the possible targets for each call are known, the location of these calls in memory as well as the location of the possible targets will vary with almost every modification of the source code, the compiler switches, new compiler versions, etc.

  Offering a mechanism that allows specification of these call targets in a manner such that the annotations do not have to be constantly changed is a difficult task. Reliability is especially an issue here, i.e. the question of whether a tool will definitely detect when the annotation is no longer valid due to changes in the code.

  Some approaches available in a commercial post link-time analyzer will be discussed below.

To summarize, ease of use and maintainability are key issues here as well as reliability. The problems regarding the former two issues mainly originate in the lack of

high-level information and the assignment between machine code and source code. The first problem is communicating the code locations that need further annotations (such annotations will be required in most applications) in a way that is comprehensible to the user. The second problem is giving the user the ability to specify these annotations in a way that is easy to write, reliable, and maintainable.

Utilization of debug information might help here, but often debug information generated by compilers can not be trusted in optimizing compilation (or many compilers refuse to generate debug information at higher optimization levels at all).

Basically such tools are nevertheless able to produce safe bounds of stack usage for many programs, as long as they reconstruct the control- and data-flow correctly. This is where the second problem arises. Analyzing machine code is very difficult. A conservative tool will typically encounter too many constructs that it does not fully understand and refuse to calculate a stack size. More aggressive tools will try to use information on typical code pattern generated by compilers to reduce these cases. This, however, is more prone to errors (as will be shown in test results later) and increases the dependency of a specific compiler version.

However, such a dependency will often be problematic as compiler vendors rarely sell older versions of compilers once a more recent version is available. When such a tool is adapted to one compiler version, this version may already be obsolete. Therefore, it is a major requirement that such tools must not silently produce wrong results if some of their assumptions about the compiler are not valid.

4.1.6 Further Related Work

The following paragraphs give an overview of work that deals in some way with obtaining bounds for stack space, but is not applicable to embedded C code. Especially, the issues of function pointers and annotations — key issues that have been addressed in this thesis — are not addressed in these works.

In the Ada [77] language, analysis of stack usage is recommended [78, 150]. Especially ADA83 was easier to analyze than C. The current ADA95 is less static and ADA subsets are usually used when static verification or calculation of stack sizes is needed [31].

Calculation of stack usage of a system in the presence of interrupts is discussed in [33], however using a formal approach rather than real code. Similarly, a first-order functional-language is examined for stack (and heap) usage in [142]. Space bounds for functional languages are also examined in [72].

Constraints that allow calculation of space (and time) bounds for recursive functions are presented in [20].

4.1.7 Example: AbsInt StackAnalyzer

When searching for actual products performing stack usage analysis for embedded software, the StackAnalyzer software from AbsInt Angewandte Informatik GmbH seems to be the most promising (and probably only) choice. AbsInt has kindly provided versions of their StackAnalyzer for PowerPC, HC12 and C16X/ST10 microcontrollers for evaluation.

This section will give a short overview about this tool. Later, results and comparisons with the stack analysis implemented for this thesis will be presented.

Usage

StackAnalyzer can be run as an interactive graphical tool as well as a command line tool without user interactions (if necessary control files and annotations are provided).

In interactive mode, a control center is shown in a window (see figure 4.1). Some configuration files as well as files containing annotations can be selected here. Also, a few options can be selected graphically. Annotations that are often needed for correct analysis have to be edited with a text editor, though.

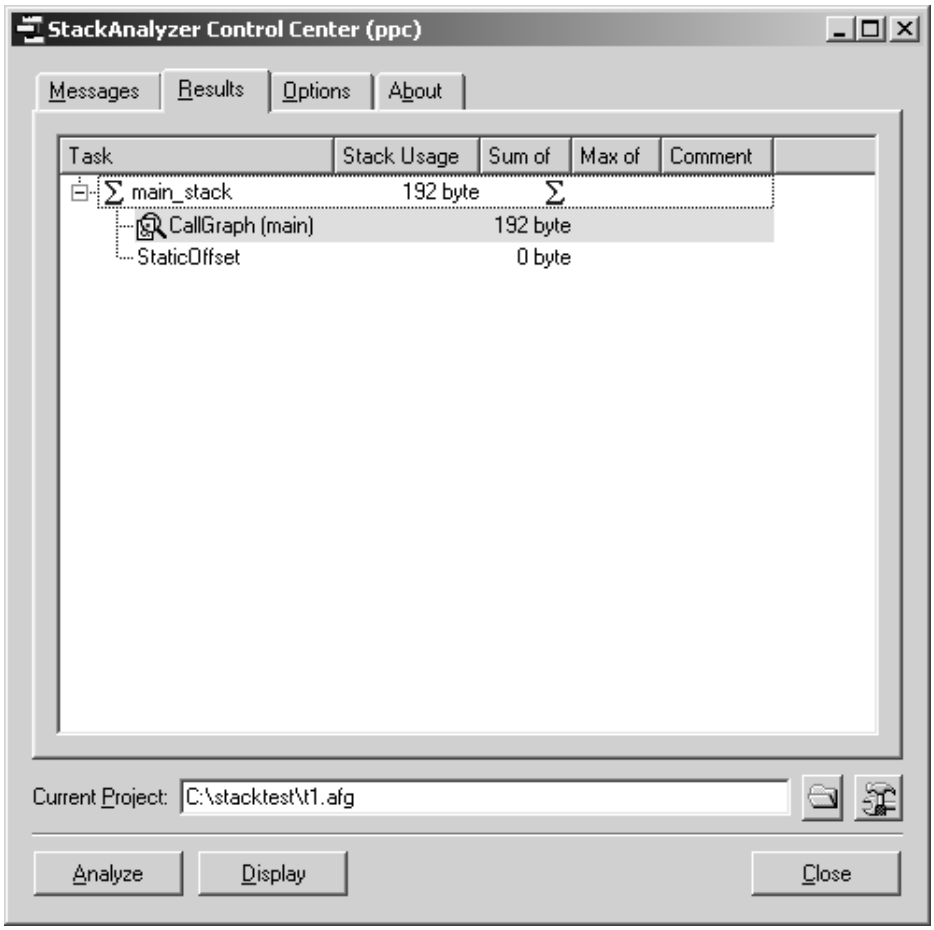


Figure 4.1: StackAnalyzer ControlCenter

The analysis performed by the StackAnalyzer always starts from a single entry point. To get upper bounds of stack usage for other entry points (e.g. other tasks), several passes are done. In interactive mode the entry point can either be selected from a list of global symbols contained in the executable or an absolute address can be specified (see figure 4.2).

Supported Input Formats

The StackAnalyzer basically falls into the category of post link-time analyzers. This is true at least for the versions for PowerPC and HC12. These tools read in absolute ELF executables for those microcontrollers which are similar to raw binary images or hex files. Additional information may be provided in ELF files, for example section attributes, global symbols, and perhaps debugging information.

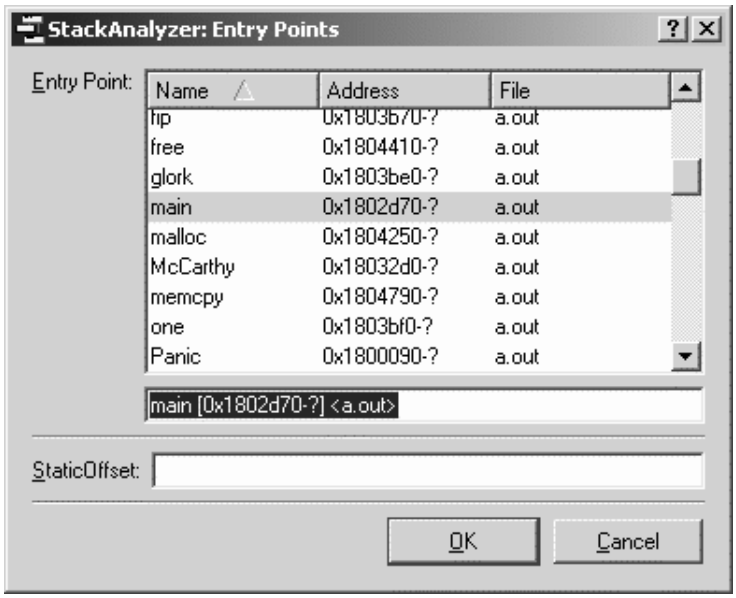


Figure 4.2: StackAnalyzer Entry Point Selection Window

An exception is the version of StackAnalyzer for the C16X/ST10 microcontroller. This one does not read linked and located images but rather assembly source files. Also, this version offers less sophisticated means for annotations and seems less polished. The versions reading linked executables seem to provide more features and less restrictions.

All StackAnalyzer versions are advertised to work with a specific version of a specific compiler. However, as they are fed the linked executables containing library code, inline assembly etc., they must not rely on code generation patterns of that specific compiler. Furthermore, compiler versions are frequently replaced by updated versions.

As long as there are no guaranteed informations about code generation of the compiler, the StackAnalyzer can not really rely on any assumptions about the generated code without actually checking it. Therefore, it is expected that the StackAnalyzer also works with other compilers producing the same output format. By recognizing some patterns of code generated by a specific compiler, however, it may be used more comfortably with that exact compiler. For example, it may recognize the jump tables generated for a switch statement only in the form that one particular compiler generates them.

Diagnostics

Diagnostics (usually error messages) are shown in a window in interactive mode or are written to a file without user interactions. Typically, these messages will be displayed if the StackAnalyzer is unable to construct a correct control-flow and call-tree or if it can not calculate the stack usage for some constructs. A list of all messages was not provided.

The locations referred to by diagnostic messages usually are displayed as addresses of machine instructions, possibly annotated with the routine containing the instruction and a disassembly of the instruction.

Annotations

If the StackAnalyzer does not understand some piece of code, for example because it can not determine the possible targets of an indirect function call, it needs annotations to continue to work. These annotations can not be made interactively but have to be written into several different text files.

The following annotations can be made to help StackAnalyzer construct a correct control-flow and call-tree:

- Memory areas can be specified to be read-only, contain only data etc.
- The stack usage of external functions can be specified.
- Limits for recursions of functions can be specified.
- Targets of computed calls or branches can be specified.

To specify targets of computed calls or branches, at first the instruction to be annotated must be identified. As mentioned above, the problem with tools dealing with binary images is the lack of the connection to the source code. Therefore, such an instruction can not be specified in terms of the source code (or — what would be even better — in the source code itself).

The easiest way to specify the instruction to be annotated is to give its memory address. This is of course supported by StackAnalyzer but imposes several problems. Pretty much every change to the program or the build process might move the instruction to another memory location and render the old annotation useless (or even seriously wrong!).

To address this problem, StackAnalyzer allows more complex expressions to specify an instruction. These can consist of a base address specified as

- an absolute address, e.g. 0x12345678,
- a label name (which must be visible in the executable), or
- a function name (which must be described in debug information)

which can be added to a combination of n units of

- instructions,
- call instructions,
- branch instructions,
- return instructions,
- instructions not altering the control flow,
- call or branch instructions with a predictable target, or
- computed call or branch instructions.

It is the last item that is probably of most importance as computed calls are the most common cases that need annotations. For example, an expression like

```
"myfunc" + 2 computed
```

refers to the second computed branch in function `myfunc`. Using such expressions, it is possible to write annotations that are not dependant on absolute addresses. However, they are still quite dangerous if the compiler changes the order of branches, a computed branch becomes a direct one (e.g. due to optimization), or an additional computed branch is added (for example because more cases are added to a switch statement and the compiler now chooses to use a jump table instead of a series of direct branches).

These forms of expressions described above are also available when specifying the targets of a computed branch or call instructions. In this case, `pc` is allowed as an additional base address. Furthermore, the case of a jump or function table (which must be in read-only memory) is supported by a special construct. It is possible to specify that a branch or call target is taken from an array that is described using the following parameters (some of which may be omitted):

- a base address (an expression like above)
- the number of elements in the array
- the size of each array element
- the offset of the address within an array element
- the size of the address within the array element
- the endianness of the address
- a mask to filter out unused bits
- a scaling factor to multiply with the extracted address
- an expression that is added (which may be `pc`)

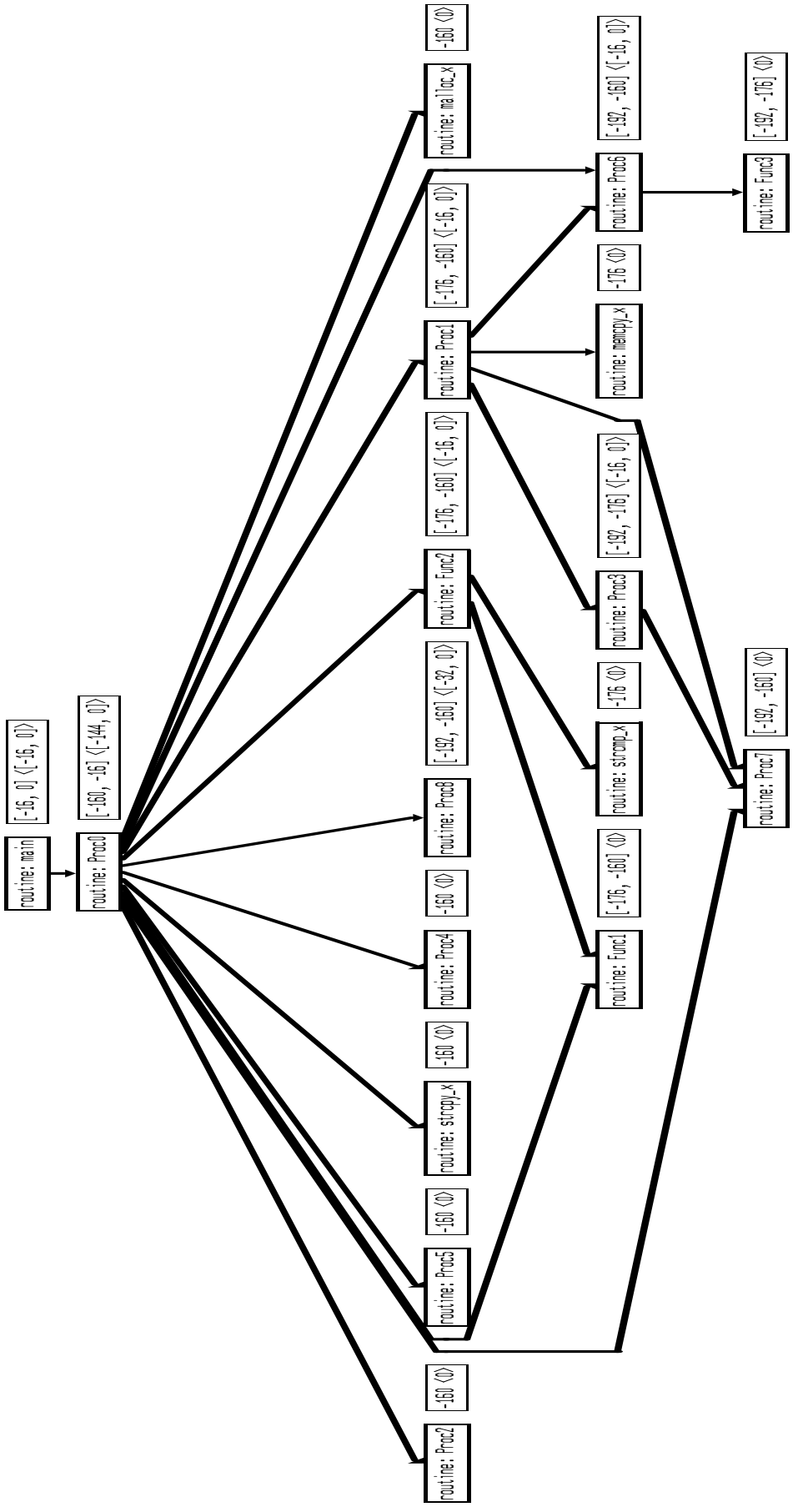
This is a very powerful method to specify indirect calls via arrays. However, it can be very difficult and time-consuming to write such array descriptors. Also, very detailed information about the layout of the array in memory is required.

Graphical Display

Apart from calculating bounds for stack usage, StackAnalyzer also is able to display a graphical view of the program analyzed (at least the part that is reachable from the specified entry point). There are different zoom options. For example, the full graph can be viewed with functions as nodes and calling relationships as edges (see figure 4.3). The results of the abstract interpretation regarding stack usage are displayed along with the nodes. For routines, the global effect on the stack pointer (i.e. including callers) is displayed first — either as an interval or as a single number if the stack modification is constant. Then the local effect on the stack pointer is displayed in the same manner surrounded by `<` and `>`. For basic blocks or single instructions, only the local effect on the stack pointer is displayed.

Further detail is shown when unfolding a function. Inside the function, the basic blocks will be shown as nodes connected by control-flow edges. The effect on the stack is now also shown per basic block.

In highest detail level, the machine instructions building a basic block can be shown together with their effect on the stack pointer. Figure 4.4 shows both levels of detail (`strcmp_x` is shown in highest detail level with individual machine instructions exposed, `Proc3` is shown at basic block level). The layout, zoom factor, etc. of the graph can be controlled using several options. The graph can be exported in several formats.



*.7

Figure 4.3: Full Graph View

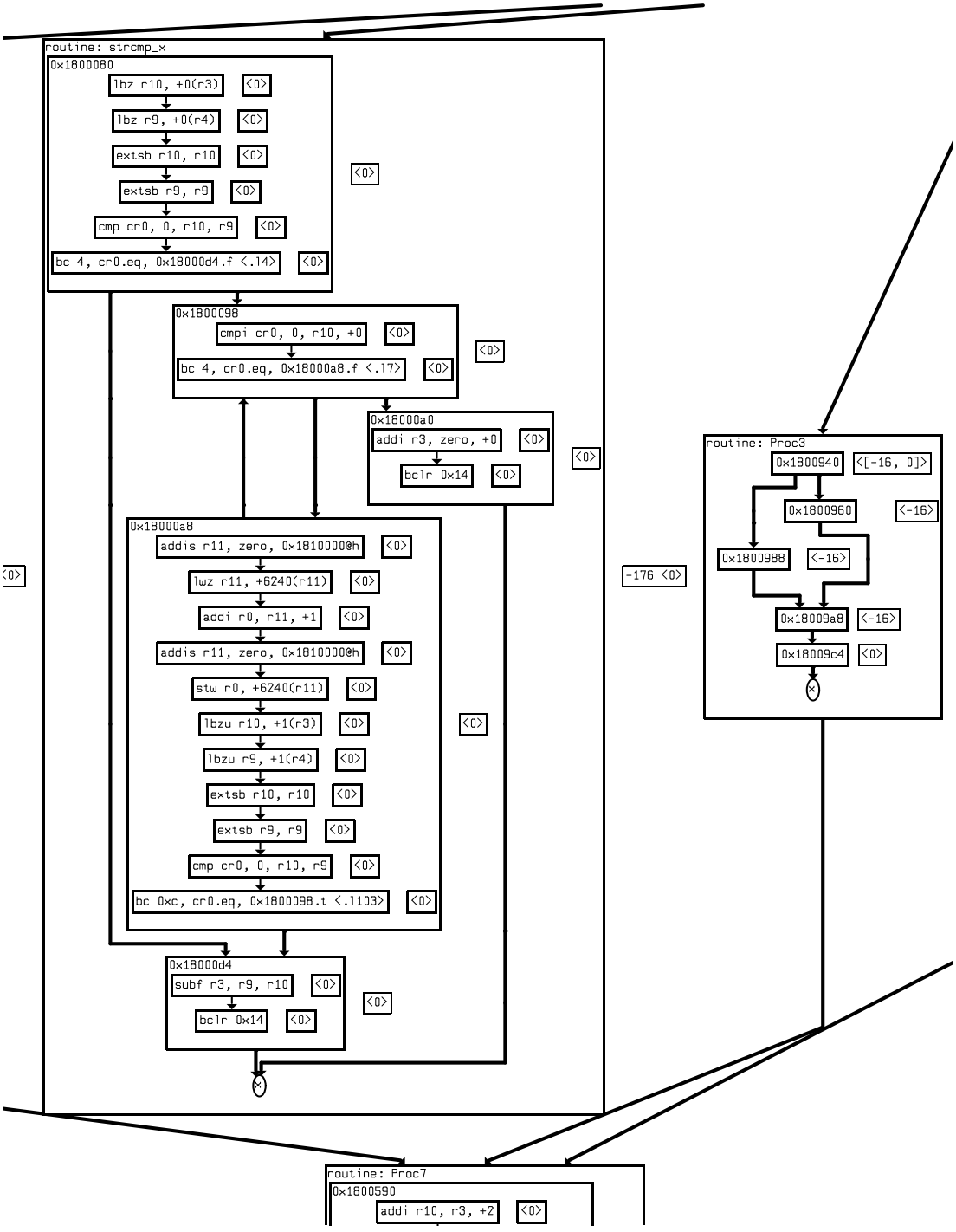


Figure 4.4: Zoom into Graph

4.2 High-Level Approach

All the approaches described above have serious shortcomings. The methods involving testing and measuring rather than static analysis always trade reliability versus efficiency. Either more memory is wasted by providing a safety-margin or the probability of failure is increased.

Careful manual analysis is very hard and expensive, but tractable if enough manpower with the required expertise is available. For the vast majority of embedded system programming that is done these days, however, it is way out of reach.

Traditional compilers do some local static analysis, have information on high-level language constructs and understand the code-patterns they generate. However, they usually only see a part of the entire application as they translate files/modules more or less separately. While modern compilers have started to perform a few optimizations like inlining across modules, stack analysis across modules does not seem to have been used so far. Furthermore, inline-assembly, library code, or indirect calls are a problem.

Post link-time analyzers seem to be the only serious attempt to address that problem so far. Real commercial products (even with special support for OSEK operating systems) are available today. By analyzing a linked binary, they have access to the entire code and are not hampered by separate compilation, inline-assembly, or library code. They do perform real static analysis and as long as they can construct a correct control- and data-flow of the system, they will provide safe and tight bounds of stack usage.

In practice, however, they are seriously handicapped by having to deal with machine-code. Information on high-level language constructs is mostly lost and binaries are hard to analyze. As a result, they will often need additional annotations that can be difficult to provide.

4.2.1 Goals

The proposed solution is to put stack analysis back into the compiler and give the compiler access to the entire source code and help it with annotations that are much easier to provide in the source code instead of in a separate file.

From now on, it is assumed that typical applications using static operating systems are used. They will use neither recursions nor dynamic memory management. Also, function pointers will be used only in a restricted fashion and usually most code (even when introduced from an external source) is provided in the form of source code (see section 2.1.8).

Only programs written in C are considered (although most of what is said holds true for similar languages like C++ or especially Embedded C++, [55]). Also, the focus is only on determining the worst-case stack usage of application tasks or interrupt service routines. It is assumed that the memory requirement of the operating system is provided otherwise and that further calculations that may be needed to obtain the final stack size required by the entire system are somehow done during the operating system generation phase. This is already the case in current systems. For the operating system, a manual analysis (perhaps aided by some tools) is tractable because it is reasonably small and the developers of the operating system have to have in-depth knowledge of the target architecture.

The following main goals should be achieved by the new implementation:

- The implementation should be a simple extension to the compiler, largely based on analysis that is already performed, mainly for optimization purposes.

- No significant penalty should be imposed on compilation times.
- The results shall be reliable. Either the analysis has to report a safe upper bound on the stack usage or it must report failure to calculate one. The only exceptions are serious bugs in the program to be analyzed. Obviously, invoking undefined behaviour (as defined in the C standard) or providing wrong annotations can not always be detected.
- Any constructs preventing a safe analysis must be diagnosed. This diagnosis should be based on C source rather than machine code if possible.
- It must be possible to use inline assembly and library functions as long as they are used in a controlled way and correct annotations are provided.
- Required annotations should be made on source code level, preferably in the source code itself. They should only have to be changed when the facts of the annotations change (e.g. when an external function is modified to use another amount of stack space. No change should be required by recompilation with new options or by changes to unrelated parts of the code.
- No intimate knowledge of the internal working of the compiler should be required to write annotations.
- Failure to compute a stack usage bound should only arise in some cases:
  - Recursive functions (not relevant here).
  - Use of inline assembly without annotation.
  - Use of an external function that is not provided as source code and is not annotated.
  - Non-trivial use of function pointers (which must not include the common case of calling a function through a constant array of function pointers or a switch statement implemented as jump table).

4.2.2 Implementation

A number of features available in vbcc are useful for this analysis and crucial in making this only a small extension. However, these are general features useful for all optimizing compilers and may well be found in many modern optimizing compilers.

- vbcc already supports a mode for cross-module optimization and allows analysis to be extended across files.
- Intra-procedural alias-analysis is already performed, although some extension was necessary (see below).
- The backends already keep track of local stack pointer modifications. This is used in most backends to address local variables directly through the stack pointer rather than requiring a separate frame pointer.
- Inline assembly as well as normal functions can have user-specified attributes.
- Jump tables for switch statements (or, in fact, series of comparison instructions) are generated by the backend rather than by the frontend.

The local stack analysis done by the backends is rather simple based on properties of the intermediate code generated by vbcc and restrictions of the C language.

One important property is that every function will reset the stack pointer (not necessarily explicitly) before returning to its caller. However, the stack pointer may move inside the function, for example if arguments are passed on the stack or variables are spilled to the stack. Such a stack pointer modification may last over several instructions. For example the following code

```
f(i)
{
 q(i);
 r(i);
}
```

is translated to the following code for the Motorola Coldfire architecture [105] (the other architectures mentioned so far all pass the first arguments in registers):

```
_f:
        move.l (4,a7),-(a7)
        jsr    _q
        move.l (8,a7),-(a7)
        jsr    _r
        addq.l #8,a7
        rts
```

One can see that the value of `i` that is pushed on the stack as argument for the call to `q` is left on the stack and removed only after the call to `r` saving one stack pointer adjustment. Also note that the parameter `i` is found with different offsets from the stack pointer (`a7`) as the stack pointer has moved but the absolute address of `i` of course stays the same.

Actually, a stack pointer modification can also extend across several basic blocks as in this example:

```
q(i)
{
 g(1, i ? x(3,4):y(5), 2);
}
```

For Coldfire it will be translated to:

```
_q:
        move.l #2,-(a7)
        tst.l  (8+l18,a7)
        beq    l3
        move.l #4,-(a7)
        move.l #3,-(a7)
        jsr    _x
        addq.l #8,a7
        bra    l4
l3
        move.l #5,-(a7)
        jsr    _y
        addq.l #4,a7
l4
```

```
move.l d0,-(a7)
move.l #1,-(a7)
jsr    _g
add.l  #12,a7
rts
```

Here it can be seen that there has been a modification made to the stack pointer that spans several basic blocks as well as additional modifications inside the blocks (i.e. the values that are pushed onto the stack as arguments for the calls to `x` and `y`). Arbitrarily deep nesting could be created.

Note however, that the additional modifications within the blocks are removed before the control-flow of these blocks joins again. This is actually the important property that makes stack analysis relatively easy in the compiler.

As a result, the only constructs that may appear in the intermediate code that can prevent calculation of stack usage are function calls and inline assembly (which, in vbcc, is represented as a special function call in the intermediate code). vbcc has to know an upper bound of stack space any called function (or inline assembly) may use (including its callees).

The data structures of functions in vbcc have been extended to be able to store two stack bounds (to accommodate architectures like the C16X that have two different stacks) as well as a flag denoting whether an upper bound of stack usage for this function is available. When a function is called, its worst case stack usage plus the current local offset to the stack pointer must be compared to the maximum stack usage encountered so far in this function to see whether a new maximum usage has been found. After all intermediate code instructions have been processed in this way, a safe upper bound for the maximum stack usage has been found.

The following reasons can be the cause of the stack size not being available at a call instruction:

- The callee has not yet been translated and therefore no stack size has been computed.
- The intermediate code call instruction refers to inline assembly rather than a function call.
- It is an indirect call through a function pointer.

External and Recursive Functions and Inline Assembly

The first problem was basically already handled by vbcc. Before optimizing and generating a function `f`, vbcc will first try to optimize and generate all functions called from `f` that have not yet been processed. This is done recursively and all functions are processed in a topological order. This works as long as the call-graph does not contain any cycles, i.e. there are no recursive function calls, and there are no external functions for which no source code is provided.

This approach is very simple and fast. It is done in vbcc to place code of called functions nearer to its callers if possible and to gather information about these functions that is later used when optimizing the caller (see section 3.3.14). For example, gen- and kill-sets for data-flow analysis can be computed more precisely if information on the callee is available. Therefore, some inter-procedural data-flow analysis can be done without any significant overhead in compilation time (basically the complexity of

optimizations is still mostly polynomial in the size of separate functions but linear in the size of the total code rather than polynomial in the entire size).

As was already mentioned in section 2.1.8, recursive functions are not to be used in applications for static operating systems. While it would be possible to implement options to specify recursion limits and still provide some stack analysis for recursive functions, this was omitted as it is not really within the scope of this thesis. Therefore, this problem only occurs with external functions here.

Also, functions that are not provided with source code are reasonably rare as even most third-party software is bought as source code. Nevertheless, the case occurs and must be correctly diagnosed and there must be a way to deal with it.

Within the compiler, the diagnosis is simple. vbcc stores the origin (i.e. file name and line of the source code) of each intermediate instruction to be able to emit useful diagnosis even during optimization or code generation phases (there are actually some intermediate instructions, e.g. created during optimization, that do not have this information but this is never the case for call instructions). Additionally, the target of a (direct) call is obviously stored in the intermediate code and also available for diagnosis.

Therefore, the call to an external function in this example

```
extern void somefunc();

int main()
{
  somefunc();
  return 0;
}
```

will cause the following diagnosis:

```
warning 317 in line 5 of "c.c": stack information for target
                            <somefunc> unavailable
```

With the information provided it should be easy to find the cause of this warning for the developer. However, there has to be a way to deal with this. If the source code for this function is available, it should simply be added to the project. If it is only provided as object code or if it is not written in C, the programmer of course will have to determine the stack size of this function somehow (for example by manual analysis — hopefully only small routines will not be available as C code).

vbcc offers a general mechanism that allows adding attributes to objects and types. This was used to implement an additional attribute (actually two for architectures with a second stack) `__stack` that can be used to specify the stack size of a function. For the following modified function an upper bound for stack usage will be computed without warning:

```
__stack(16) extern void somefunc();

int main()
{
  somefunc();
  return 0;
}
```

As inline assembly is handled internally pretty much like a function call in vbcc, what was said about external functions above holds true for inline assembly in the same

way. The diagnostics are just as precise and the new `__stack` attribute is available to specify the maximum stack usage of inline assembly:

```
__stack(0) double sin(__reg("fp0") x)="␣fsin␣fp0";

int main()
{
  double one = fsin(3.14159 / 2);
  return 0;
}
```

Targets of Computed Calls

The remaining (and larger) problem is computed calls that may also be contained in the intermediate code. In this case, the operand of the call instruction is a dereferenced function pointer variable. As vbcc already does intra-procedural context-sensitive alias analysis, this data is used.

First, intermediate call instructions have been extended to store a list of possible targets when/if they can be computed. During the alias analysis phase, vbcc tries to compute a set of targets that any pointer may refer to. If a call instruction through a function pointer variable is found, the possible functions the pointer may refer to, are used as a list of targets for this call. For example, in this code

```
extern void f(), g(), h();

int main(int i)
{
  void (*fp)();
  if(i)
    fp = f;
  else
    fp = g;
  fp();
}
```

vbcc can detect that the call through `fp` will either be a call to `f` or to `g`. vbcc performs context-sensitive alias analysis, i.e. in this example

```
extern void f(), g(), h();

int main(int i)
{
  void (*fp)();
  if(i)
    fp = f;
  else
    fp = g;
  fp();
  if(i > 3)
    fp = h;
  fp();
}
```

it will detect that the first call through `fp` will still either be a call to `f` or to `g` whereas the second call could reach `f`, `g` or `h`. The maximum stack usage of all possible targets is then used for computation of the stack size.

However, there are the usual limitations on data-flow analysis. If, for example, function pointers are passed as function arguments, or global function pointer variables are used, vbcc often will not be able to compute a safe list of all possible targets. In the presence of these constructs, analysis is rarely possible.

As such constructs are not expected in applications for static operating systems (see section 2.1.8), no further support has been implemented. These calls will, however, be correctly diagnosed and no bogus stack size will be emitted. The small example

```
int main(void (*fp)())
{
  fp();
}
```

will produce the diagnostic:

```
warning 320 in line 3 of "c.c": unable to compute call targets
```

It would of course be possible to provide a way to manually specify the targets of such a call, but this was not considered of much importance. In the case that such constructs are used it is even doubtful whether any programmer can reliably denote all possible targets — it surely would be extremely difficult to maintain correctly during development.

Arrays of Function Pointers

While most of theses cases can indeed be ignored when dealing with applications using static operating systems, some special cases are used sometimes nevertheless. Notably, as was mentioned before, function tables in constant arrays are used. Fortunately, the C language allows the declaration of such an array as `const` and developers will usually declare them correctly as otherwise the array would be put into the more expensive RAM rather than into ROM.

The alias analysis phase of vbcc had to be extended a bit to handle these cases. As the initializers of constant arrays are usually available, vbcc can extract all function targets from such arrays and use them for further analysis. In this example

```
extern void f(), g(), h();
void (*const (fa[]))()={f, g};

main(int i)
{
  fa[i]();
}
```

vbcc can detect automatically that only `f` and `g` can be called. No annotations are needed. The targets obtained from array initializers are also used in further alias analysis. Therefore, in the following example

```
extern void f(), g(), h(), q();
void (*const (fa[]))()={f, g};
void (*const (fb[]))()={q};

main(int i, int j)
```

```
{
  void (*fp)();
  if(j)
   fp = fa[i];
  else
   fp = fb[i];
  fp();
}
```

vbcc detects that the call can reach `f`, `g` or `q`, but not `h`. Function pointers may also be included in arrays of structures etc. However, all functions in the initializer will be assumed to be possible targets. Computing somewhat more context-sensitive information would be possible but not without some effort.

As has been mentioned above, the stack analysis built into vbcc relies on callees being handled before their callers. To make this work also for computed calls, some further additions had to be made.

If, for a computed call, all possible targets have been determined, but there is no stack information for one or more targets, a warning is emitted. This diagnostic is the same as that for a single function which was described above. Therefore, a developer can easily find and annotate not only the function without stack information but also the line of code where it may be called.

4.2.3 Results and Comparison

To verify the effectiveness and correctness of the new stack analysis, a few test programs were analyzed and the results have been compared with the ones produced by the AbsInt StackAnalyzer that was described above. Several test programs will be discussed and analyzed with both tools. The results of the analysis as well as the problems to obtain them will be analyzed.

Levels of Context-Sensitivity

During this test phase it was soon obvious that the actual upper limits for the stack usage produced by both tools was always identical (with one notable exception that will be mentioned below). This suggests that both tools provide the same level of context-sensitivity. Tools computing the maximum stack usage only from a call tree and local per-function stack usage would, for example, obtain less precise bounds. In this code

```
int main()
{
  f(1,2);
  g(1,2,3,4);
  return 0;
}
```

the call to `g` might happen at a point with higher stack usage (as more arguments have been pushed — depending on the target architecture). A context-insensitive tool would use the maximum local stack offset (i.e. the instruction calling `g`) plus the maximum stack usage of all functions called (which might be `f`). Therefore, it would calculate a stack usage that can, in fact, never be reached as `f` is not called at the point of maximum local stack usage. StackAnalyzer as well as vbcc both take this fact into account.

On the other hand, some very sophisticated analysis could be imagined. In this example

```
extern void a(), b(), c(), d();
void (*const (fa[]))() = {a, b, c, d};

int main(int i)
{
  fa[2 * i]();
  return 0;
}
```

only `a` and `c` are possible call targets. An analyzer that knows this might be able to provide a tighter bound for the stack usage. Neither StackAnalyzer nor vbcc (or, in fact, any other tool known to the author) do this kind of analysis.

Therefore, the main focus in these tests will not be on the stack bounds that have been computed but rather on how many annotations have been necessary to obtain the stack size upper bounds and on how these annotations are handled.

Test Environment

AbsInt has kindly provided three versions of their StackAnalyzer. Versions for the PowerPC and HC12 which read ELF executables as input as well as a version for the C16X/ST10 that reads assembly sources in the format of the Tasking assembler. Although specific commercial compilers are recommended by AbsInt to be used with StackAnalyzer, the goal was to use StackAnalyzer on code generated by vbcc to be able to compare the results.

There is a full PowerPC backend for vbcc that can produce code in ELF format as well as a reasonably working backend for the C16X that can also produce assembly output in Tasking format (although this is only an option and has some restrictions — another output format is the default for vbcc). For the HC12 there is an incomplete backend that produces output in ELF format, but it is still rather limited and can only translate a subset of C.

As a result, most tests were done with PowerPC code. There is a common ABI [122] that is adhered to by vbcc as well as the compiler (DiabData 4.4b) recommended by AbsInt. There are also no unexpected exotic code patterns generated, so the code generated should pose no big obstacles — various versions of the DiabData compiler might differ just as much. To verify how much StackAnalyzer benefits from code-patterns it knows from the DiabData compiler, most tests have been run with this compiler also.

The ABI used for the PowerPC has some restrictions regarding testing stack analysis features. Every function will allocate all stack it needs during the function prologue. All stack slots required for temporaries or function arguments are allocated in advance with a single instruction. Therefore, the example for context-insensitivity that was presented above, would have no meaning in this ABI.

To verify this feature, some of the tests have been performed also for the C16X and HC12 architectures. However, the StackAnalyzer for the C16X reads some of its annotations from comments in the assembly file rather than from external files. The Tasking compiler that is suggested to be used with StackAnalyzer for C16X, offers the possibility to write the C code as comments into the assembly output and, using this option, it is possible to write annotations as comments into C source. vbcc, on the other hand, does not provide this option (in an optimizing compiler there is no

reasonably simple mapping between source code and machine code). Therefore, test cases requiring annotations for the StackAnalyzer have not been run for the C16X.

Wherever a test case was using standard library functions, these have been annotated for both tools where necessary. Function inlining and loop unrolling was always turned off for the tests to prevent elimination of the stack analysis problem in some cases.

Test Cases

The following test cases were used to validate and compare both tools. Some are synthetic benchmarks that will test certain features of the analysis whereas others are existing examples to show how the tools handle larger pieces of real code. However, the code has been chosen such that it mostly meets the requirements for small embedded applications, i.e. no recursions or dynamic memory management.

All the synthetic benchmarks have been compiled together with the following extra module that provides some functions with increasing and easily identifiable stack-usage:

```
f()
{
  int a[10],i,s;
  for(i=0;i<10;i++)
    a[i]=i;
  for(s=0,i=10-1;i>=0;i--)
    s+=a[i];
  return s;
}

g()
{
  int a[100],i,s;
  /* analogue */
}

h()
{
  int a[1000],i,s;
  /* analogue */
}
```

The following synthetic test cases were used:

- **t1.c**

  This is the most simple test case that was used. Just a function using some stack space and calling a function that also uses some stack:

  ```
  int main()
  {
   int a[10];
   a[9]=0;
   return f()+a[9];
  }
  ```

- **t2.c**

  This test case uses some inline assembly that does not use the stack. It is used to test how simple code not generated by a C compiler is handled.

```
__stack(0) void asm_cee() = "\tmtspr\t80,0";

int main()
{
  asm_cee();
}
```

- **t3.c**

  This one tests how complicated assembly code that is hard to analyze is handled. A small loop pushes some bytes on the stack that are afterwards cleaned up by an addition to the stack pointer.

```
__stack(20) void asm_push20() =
  "␣li␣3,5\n"
  "lab:\n"
  "␣stwu␣0,-4(1)\n"
  "␣addic.␣3,3,-1\n"
  "␣bne␣lab\n"
  "␣addi␣1,1,20";

int main()
{
  asm_push20();
}
```

- **t4.c**

  A large switch statement is the main part of this test case. It will be translated into a jump table by many compilers.

```
int main(int i)
{
  switch(i){
  case 1:
    f();
  case 2:
    f();
  case 3:
    f();
  case 4:
    f();
  case 5:
    f();
  case 6:
    f();
  case 7:
    g();
  case 8:
```

```
    f();
  case 9:
    f();
  case 10:
    f();
  case 11:
    f();
  case 12:
    f();
  case 13:
    f();
  }
}
```

- **t5.c**

  The reasonably common case of a constant array of function pointers is tested here. It is basically the most simple case of such a construct.

```
int f(), g(), h();

int (*const (fa[]))()={f, g};

int main(int i)
{
  return fa[i]();
}
```

- **t6.c**

  The previous test case is made a bit more complicated as there is an array of constant structures that contain not only function pointers but also normal numbers.

```
int f(), g(), h();

const struct {char f;int (*fp)();} fa[2]={1, f, 2, g};

int main(int i)
{
  return fa[i].fp();
}
```

- **t7.c**

  To check whether some context-sensitive analysis is done, this test case is used. On architectures that push arguments on the stack during a function call, it will call the functions `g` and `f` at different stack levels. A context-insensitive tool might compute a larger stack size than necessary.

  This test case does not make sense on the PowerPC as the compiler allocates all stack slots for arguments in the function prologue. It is useful on the C16X, however.

```
int main()
{
  return q(1,2,3,4,5,f(),g());
}
```

- **t8.c**

  Limited local handling of function pointer variables is tested with this example. Some data-flow analysis is necessary to handle the case without annotations.

```
int f(), g(), h();

int main(int i)
{
  int (*fp)();
  if(i)
    fp=h;
  else
    fp=g;
  return fp();
}
```

- **t9.c**

  A bigger structure is passed as a function argument in this test case. Some compilers might emit code that pushes data on the stack in a loop. However, none of the compilers used in this test, did that (they all just moved the stack pointer with one instruction before copying the argument). vbcc for the 68k/Coldfire architecture would have done it in this case, but there is no StackAnalyzer for this architecture.

```
struct {int a[40];} s;

main()
{
  f(s);
}
```

Additionally, some larger source have been used as test cases.

- **dry2_1.c**

  This is a somewhat modified version of the famous Dhrystone benchmark by R. Weicker. It is distributed as an example with StackAnalyzer. The source file is 12204 bytes long and consists of 452 lines of C code.

- **cq.c**

  This is large test program that verifies a C compiler for compliance to the C standard (more or less). It is distributed as test case with the lcc C compiler [60]. It calls its test routines in a loop through function pointers taken from an array.

  The following changes have been made to the code in order to make it resemble an embedded program:

- The output has been eliminated to avoid having to annotate several system calls.
- One small recursion has been removed.
- The array of function pointers has been declared constant.

This source file is 124473 bytes long and consists of 5345 lines of C source (including comments).

Results with PowerPC Code produced by vbcc

Table 4.1 shows the results of stack analysis on the test cases performed by vbcc as well as by StackAnalyzer. As was already mentioned, the values of stack sizes are almost always the same. Therefore, the major criteria for comparison is the number of annotations needed. The less annotations it needs, the more usable a tool is. S_v denotes the stack size calculated by vbcc, S_s the stack size computed by StackAnalyzer, A_v the number of annotations needed by vbcc and A_s the number of annotations required by StackAnalyzer.

Table 4.1: Stack Analysis Results PowerPC

test case	S_v	S_s	A_v	A_s	Remarks
t1.c	56	56	0	0	
t2.c	8	8	**1**	0	see 1)
t3.c	28	28	1	1	see 1)
t4.c	416	416	0	**1**	see 2)
t5.c	424	424	0	**1**	see 2)
t6.c	424	424	0	**1**	see 2)
t7.c	424	424	0	0	
t8.c	4016	416	0	0	bug in StackAnalyzer, see 3)
t9.c	216	216	0	0	
dry2_1.c	152	152	0	0	
cq.c	576	576	0	**1/n**	see 2), 4)

1. When there is inline-assembly included in a C file, both tools behave quite differently:

   - vbcc does not understand the inline-assembly at all and always requires an annotation that can be written as a simple attribute to the code.
   - StackAnalyzer can understand most "simple" assembly constructs and therefore does not need any annotation in many cases — in such cases it has an advantage. If, however, it does not understand the code like in `t3.c`, it can not be annotated without re-writing the code.

     While it is possible to specify the stack-usage of a routine to StackAnalyzer, this can not be done for inlined code.

   It is not clear which approach is the better solution here. The annotation required by vbcc is usually easy to create for the person writing the code. Only few instructions will typically be written as assembly code — complex code involving control structures etc. should be written in C anyway.

   Nevertheless, StackAnalyzer does not require any annotations for such code in most cases and therefore is still easier to use. The cases where StackAnalyzer can

not cope with the inline assembly are probably rare in normal application code. However, if such a case occurs, StackAnalyzer causes much bigger problems as the code has to be re-written or the stack usage of the entire function containing the inline-assembly must be calculated manually and specified rather than specifying just the stack usage of the assembly code itself.

2. In all the test cases involving tables of function pointers or jump tables produced by switch statements, StackAnalyzer needed annotations. There are some problems with these annotations:

   - Locating the corresponding construct in the source code is not easy. StackAnalyzer will only display the function containing the construct (which can be wrong if the code was inlined) and the address of the machine instruction representing the computed call or branch.

     Also, specifying this instruction is somewhat dangerous as was described above — even though StackAnalyzer certainly provides some sophisticated methods to address such instructions (see section 4.1.7).

   - To specify the array containing the target addresses, the developer must have intimate knowledge of the layout of these tables in memory. Not only pointer sizes, but also alignments of different types may have to be known.

     In the case of the switch table or static function tables (as used in `cq.c`), there are additional problems. The jump table will not have a global name and must therefore be addressed by its address. This is, of course, hard to find and subject to change with every new build of the system. Also, the developer must know whether the entries are absolute, PC-relative, etc. It is unlikely that a switch statement that is not recognized by StackAnalyzer can be annotated in a useful and maintainable way.

3. This test case was incorrectly handled by StackAnalyzer. It tried to determine the possible values of the function pointer, but, probably due to wrong/no data-flow analysis, only one of the two possible targets of the computed call was recognized. Therefore, the call-tree and the stack-usage was not computed correctly.

   As a result, an incorrect (too small) stack size was displayed without any warning, i.e. the user would not know that StackAnalyzer was unable to determine the correct size. Figure 4.5 shows the graph output of StackAnalyzer. Although it recognizes that there are two paths leading to the indirect call instruction (`bclrl`), it assumes that always `g` will be called and ignores the possible call to `h`. Note that only an edge to routine `g` is included, but not to `h`. AbsInt announced an update that fixes that bug but an annotation will be required.

4. `cq.c` contains an indirect call that always jumps to the same function:

   ```
   pfi = glork;
   if((*pfi)(4) != 4){
     ...
   }
   ```

   vbcc recognizes this and optimizes it into a direct call. If this optimization is turned off, StackAnalyzer will need an additional annotation for this call.

   Furthermore, if loop-unrolling was not turned off, the main loop that calls the separate parts of the program via a function pointer array would have been un-

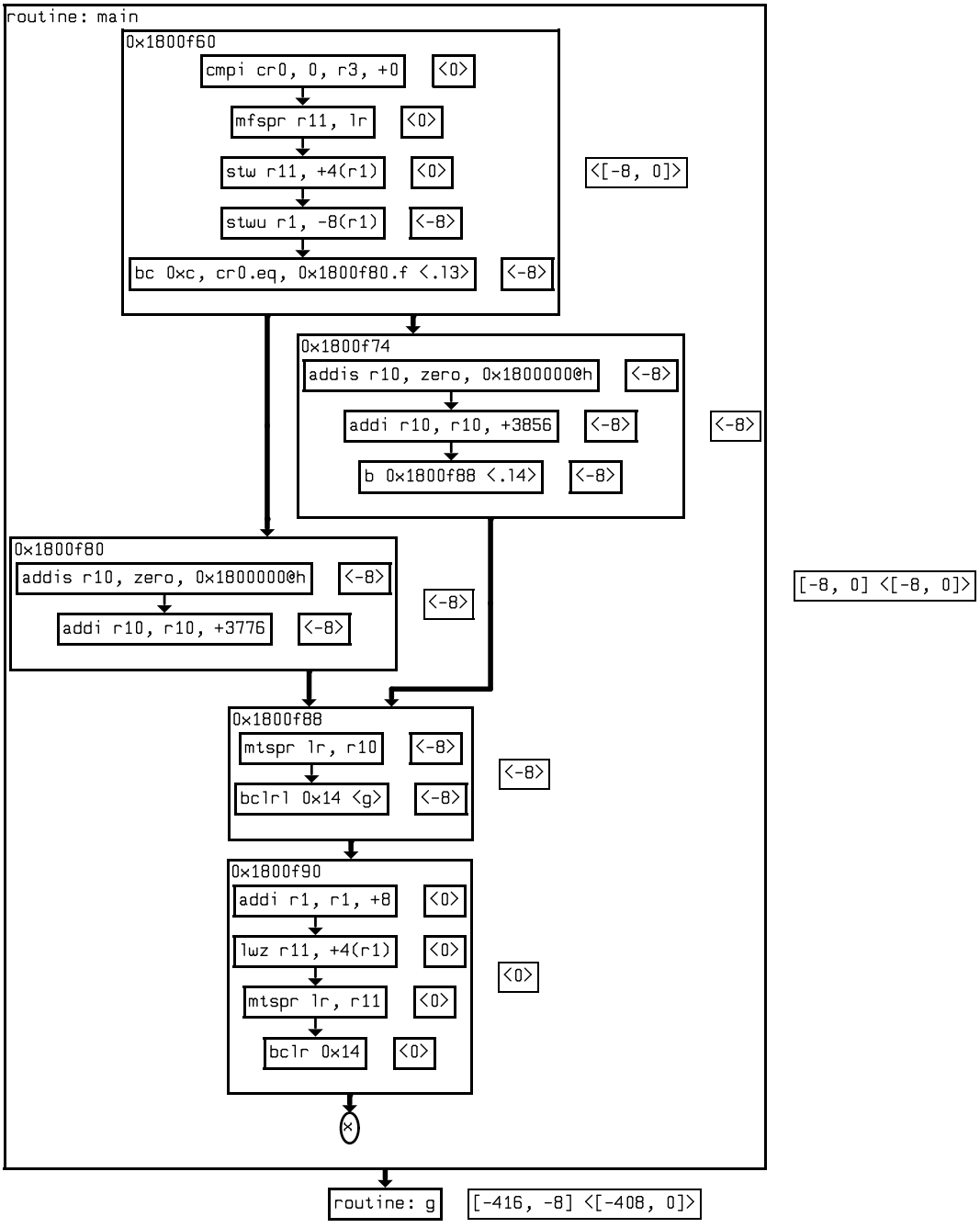


Figure 4.5: StackAnalyzer computes wrong stack

rolled. In this case, there would have been several instances of this computed call that would have had to be annotated when using StackAnalyzer.

Results with HC12 Code produced by vbcc

The small test cases have also been run with the HC12 versions of vbcc and StackAnalyzer. Basically, there were no significant changes to the results from the PowerPC. One exception is that the HC12 backend of vbcc does not emit jump tables for switch statements at the moment (which helps StackAnalyzer to correctly understand `t4.c`). Additionally the HC12 version of StackAnalyzer does not emit bogus results for `t8.c` but rather displays an error message and requires an annotation.

`t7.c` shows that StackAnalyzer does context-sensitive analysis. Table 4.2 shows a list of the results from HC12 that differ somehow from the PowerPC results:

Table 4.2: Stack Analysis Results HC12

test case	S_v	S_s	A_v	A_s	Remarks
t4.c	208	208	0	0	no jump table generated
t7.c	208	208	0	0	
t8.c	2008	2008	0	**1**	annotation required by StackAnalyzer

Results with C16X Code produced by vbcc

Finally, the small tests were also run using the C16X versions of vbcc and StackAnalyzer. One point that makes the C16X different is its use of two separate stacks. It has a small hardware stack that is supported by special instructions (e.g. a `call` instruction that pushes the return address on this stack). However, this stack is very limited in internal memory and not sufficient for many programs. Therefore, the most common memory model used on this chip uses the hardware stack for return addresses only and uses a software stack (accessed through a general purpose register) for local variables and function arguments.

Additionally, (in the memory model used) function pointers are 32bit whereas the register size is only 16bit. Therefore, function pointers have to be kept in a register pair and there is no machine instruction performing an indirect call but some instruction sequence is needed that first pushes both parts of the return address, then pushes both parts of the function pointer and finally executes a return from subroutine instruction.

Most tests gave the expected results. As was the case for the HC12, the C16X backend of vbcc does not emit jump tables. Therefore, StackAnalyzer did not need annotations for `t4.c`. `t7.c` showed that also the C16X version of StackAnalyzer does context-sensitive analysis. The results from the C16X are shown in table 4.3.

Table 4.3: Stack Analysis Results C16X

test case	S_v	S_s	A_v	A_s	Remarks
t4.c	200/4	200/4	0	0	no jump table generated
t7.c	200/4	200/4	0	0	
t8.c	2000/8	**0/8**	0	**1**	warning displayed by StackAnalyzer

Again, test case `t8.c` is a problem for the StackAnalyzer. vbcc generates the following code for the indirect call:

```
        mov    R11,#SEG l69
        push   R11
        mov    R11,#SOF l69
        push   R11
        push   R15
        push   R14
        rets
l69:
        rets
```

The function pointer is contained in registers R14 and R15. The first two `push` instructions will push the return address (label `l69`) and the third and fourth will push the address to be called. The `rets` is a typical return-from-subroutine instruction that pops an address from the stack and jumps there. In this case, it is used to implement an indirect jump rather than a return — a common technique.

At the callee, the address of label `l69` will be at the top of the stack and therefore the callee will return to this address. The second `rets` instruction is the actual return from function `main`. As can be seen in the graph output of StackAnalyzer, it does not understand that mechanism and considers the first `rets` as the function exit and the second one as unreachable (see figure 4.6). It will display a wrong stack size, but at least a warning is displayed, because StackAnalyzer thinks that the function pushes more bytes on the stack than it removes.

Using Recommended Compilers

The tests performed so far have been with code compiled by vbcc. However, other compilers are recommended for StackAnalyzer, and StackAnalyzer has been optimized to better understand the code generated by them. Therefore, additional tests have been made with code compiled by those compilers (and containing debug information where possible). Naturally, the stack size can not be compared with the results of vbcc as the code may differ from the code generated by vbcc.

The following items have been tested:

1. Are the diagnostics better? For example, is the source line of a computed call displayed if it has to be annotated?
2. Is StackAnalyzer able to compute the list of possible targets in simple calls through constant arrays, i.e. does it handle `t5.c` without annotations?
3. Is the problem that produces a wrong stack size for `t8.c` on the PowerPC solved?
4. Are switch tables recognized without annotations?

The answers to the first three items are all “no”, i.e. there is no improvement if the recommended compilers are used. The diagnostics are not better, the same annotations are necessary for function tables and the bug with `t8.c` is still there.

The last item, however, has been improved. Apparently StackAnalyzer recognizes typical code-patterns that are emitted for switch statements by those compilers. On all three architectures `t4.c` was correctly analyzed without annotations. Furthermore, AbsInt provided a small example containing function pointer tables that were automatically detected by StackAnalyzer. However, the code-pattern was recognized only with certain compiler-switches. For example, changing optimization options produced slightly different code which was not recognized anymore. Some other problems will be described below.

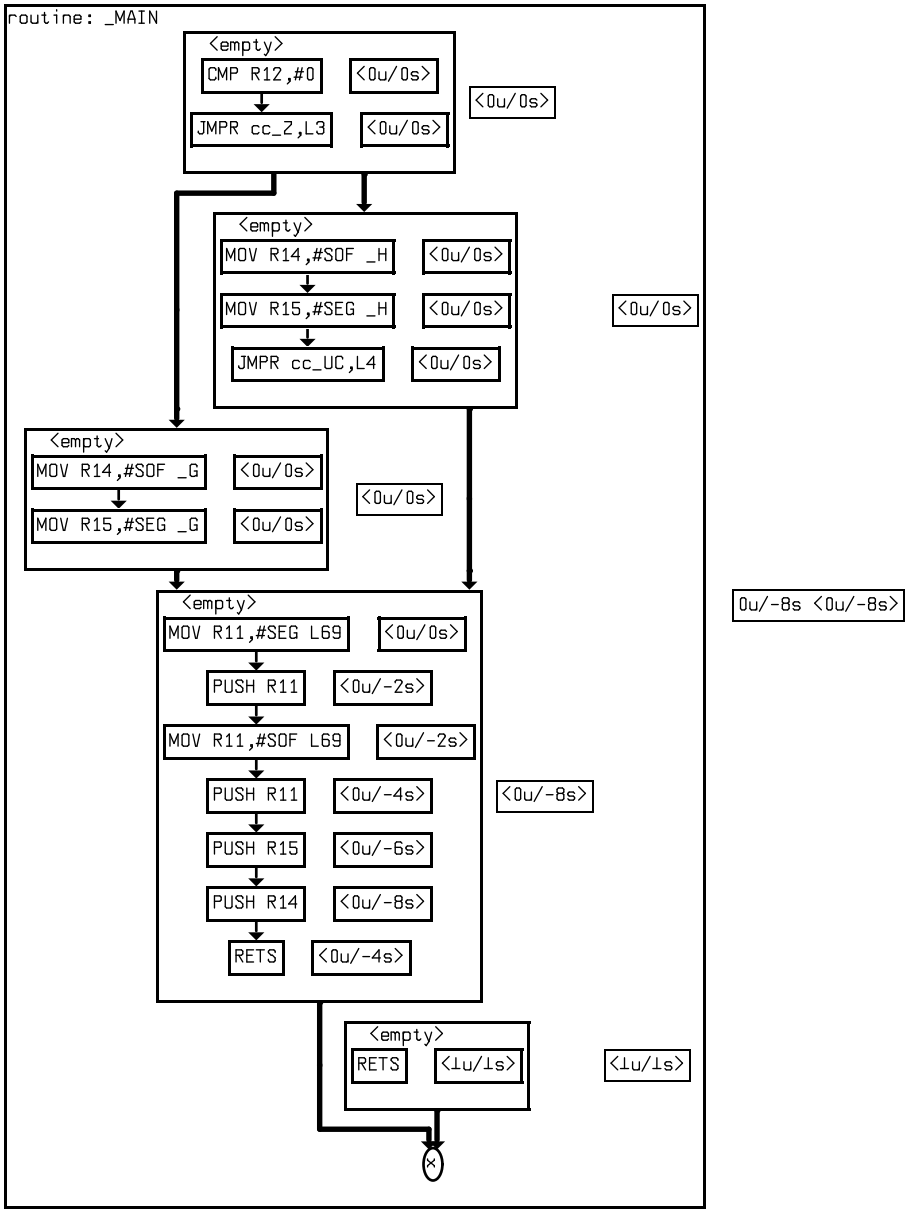


Figure 4.6: t8.c interpretation by StackAnalyzer

Susceptibility of Pattern-Matching

Now there is the question of whether the pattern-matching (or whatever algorithm) used by StackAnalyzer works safely or whether it can be tricked into yielding false results. Generally, when recognizing such code-patterns there is a trade-off between hit-rate and safety (i.e. no false positives).

First, the C16X version was tested, as it reads assembly source directly and was the easiest to handle. After a few attempts, the following source was soon produced which tricked StackAnalyzer into giving false results:

```
T4_2_CO SECTION LDAT WORD PUBLIC 'CROM'
__swtab LABEL  WORD
       DW     SOF _3
       DW     SOF _3
       DW     SOF _4
       DW     SOF _4
T4_2_CO ENDS

T4_1_PR SECTION CODE WORD PUBLIC 'CPROGRAM'
       PUBLIC _main
_main  PROC   FAR
       mov    r12,#3
       jmp    _tricky
       SUB    R12,#01h
       CMP    R12,#01h
       JMPR   cc_UGT,_3
_tricky:
       SHL    R12,#01h
       ADD    R12,#__swtab
       MOV    R12,[R12]
       JMPI   cc_UC,[R12]
_4:
       CALLS  SEG _h,_h
       rets
_3:
       rets
_main  ENDP
T4_1_PR ENDS
```

Basically, all that had to be done was to duplicate the code-pattern emitted by the Tasking compiler for a switch-statement with two cases and insert a jump around the range check (the part of code emitted for a switch-statement that verifies that the argument is within the range of the case labels). StackAnalyzer ignores the possible branch that skips the test and thinks that only the first two addresses in the jump table can be used (which would be true if the range check was not skipped), whereas in reality the value of the argument has been chosen to always use the fourth one.

The graph output in figure 4.7 shows the wrong interpretation of this code by StackAnalyzer. Note that there is no edge leading to the block that calls `h`.

A similar attempt was made using the PowerPC version. This was a bit harder as the code-pattern of the DiabData compiler is a bit more complicated. Also, the StackAnalyzer for the PowerPC did not recognize the code as switch statement if there was an unconditional branch behind the range check. However, using a conditional

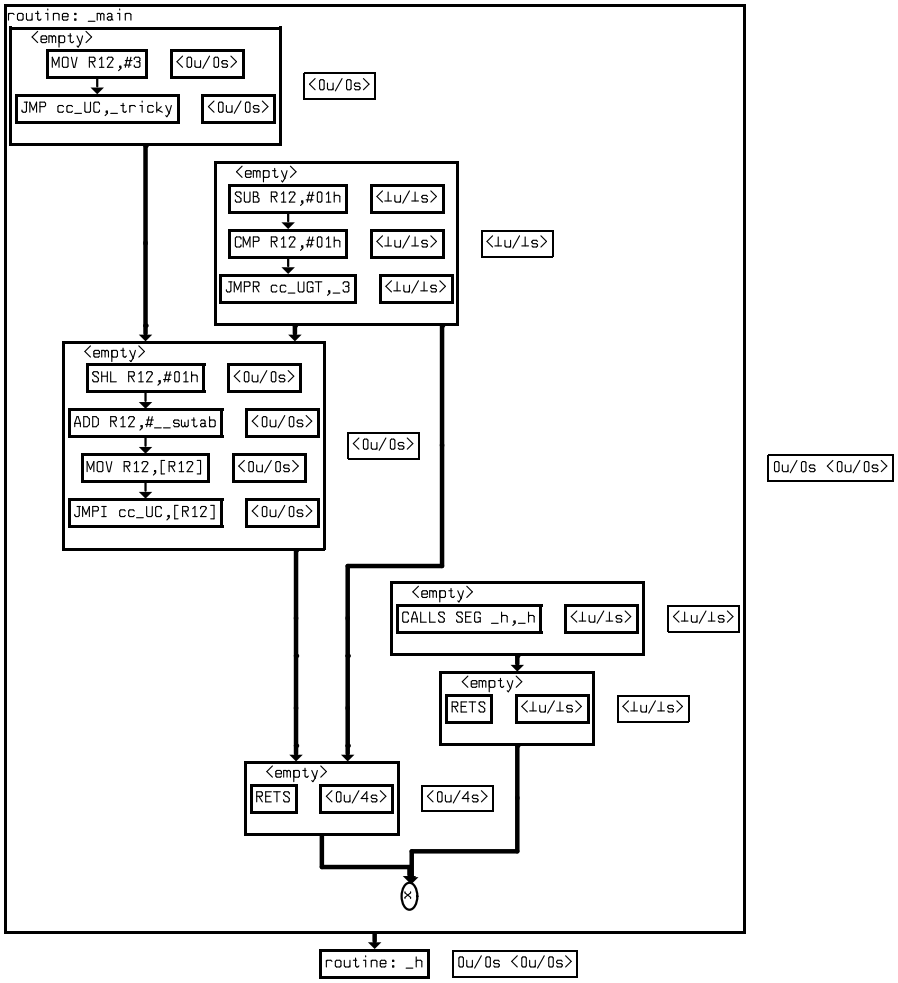


Figure 4.7: Faked Switch Statement on C16X

jump with a condition that is always true, solved that problem and the following code fools StackAnalyzer into producing incorrect stack usage information:

```
        .text
swtab:
        .long  l3
        .long  l3
        .long  l4
        .long  l4
main:
        mflr   0
        stwu   1,-8(1)
        stw    0,12(1)
        li     3,3
        slwi   11,3,2
        addis  12,11,swtab@ha
        cmpwi  3,0
        bgt    tricky
        cmplwi 3,1
        slwi   11,3,2
        addis  12,11,swtab@ha
        bgt    0,l3
tricky:
        lwz    11,swtab@l(12)
        mtctr  11
        bctr
l4:
        bl     h
l3:
        lwz    0,12(1)
        mtlr   0
        addi   1,1,8
        blr
```

The graph output of StackAnalyzer in figure 4.8 shows the wrong control-flow that is produced. Note that the block that calls `h` is not even displayed in the graph as it is not considered reachable by StackAnalyzer. This can be verified when looking at the addresses of the basic blocks. The block containing the indirect branch starts at address 0x1800eb0 and contains three instructions (which are all four bytes on the PowerPC). However, the next block starts at address 0x1800ec0 rather than 0x1800ebc. This leaves a gap for exactly one instruction — the call to `h`.

No attempt has been made to trick the HC12 version as the recommended compiler usually emits library code for switch-statements rather than inline code.

It can be argued that this code is artificially constructed and will not be created by compilers or appear in normal applications. However, unless that is proven, there is a certain level of uncertainty included in the results. Presumably these problems are simply bugs that could be fixed. It should be possible to reliably detect certain patterns implementing a switch statement. However, problems are likely to arise if the compiler varies the code produced for a switch statement in a non-trivial way. While it would still be possible to avoid false positives, annotations would be required in those cases. In any case, there is a dependency on the compiler that has generated the code

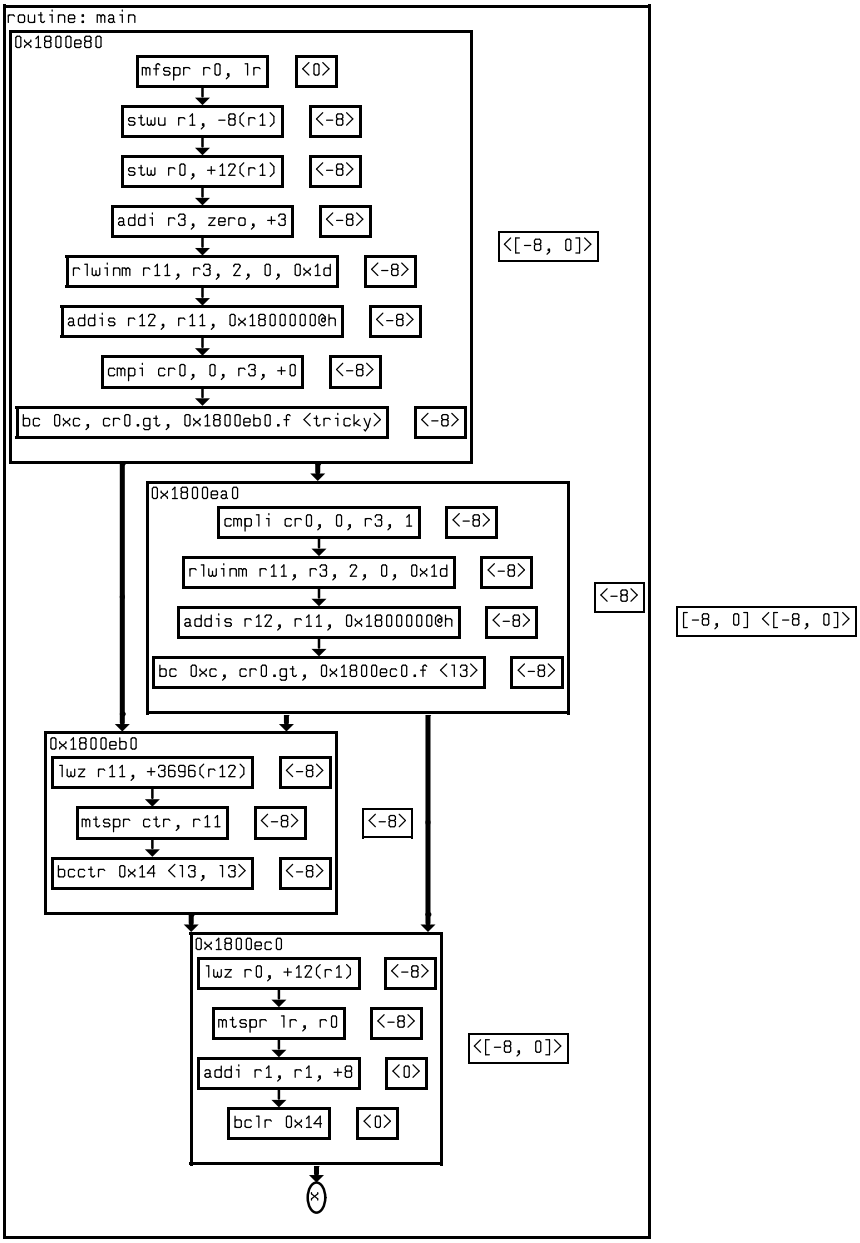


Figure 4.8: Faked Switch Statement on PowerPC

that is to be analyzed.

Finally, the example that was provided by AbsInt and automatically extracted the call targets from an array was analyzed. Basically, the following code translated from the C statement `arr2[i]();` was recognized:

```
addis          r12,r0,arr2@ha
addi           r12,r12,arr2@l
rlwinm         r11,r31,2,0,29
lwzx           r12,r12,r11
mtspr          lr,r12
blrl
```

The first two lines load the address of the array into a register. The third instruction multiplies the index by four (size of a pointer), the fourth loads the pointer from the array and the last instructions perform the actual indirect jump. The StackAnalyzer only sees the binary code, however. Therefore, it can recognize only that the code jumps to a location that is loaded from `arr2` plus some index multiplied by four.

To obtain the possible call targets, StackAnalyzer apparently makes use of the fact that ELF executables can still contain symbols and sizes of objects. It can check that the base address loaded in this example corresponds to the symbol `arr2` and that the object starting at this address has a size of 48 bytes (the array in the example has 12 entries). At first, it seems this is a reasonable approach as long as the symbol information is there.

Unfortunately, the executable does not show that the compiler really wants to load the address of `arr2` at this point. The example provided by AbsInt also contained another function table `arr1` with 12 entries that was located in front of the second array. Changing the call

```
arr2 [i]();
```

to

```
i += 12;
arr2 [i - 12]();
```

produced the following machine code for the indirect call:

```
addis          r12,r0,arr2-48@ha
addi           r12,r12,arr2-48@l
rlwinm         r11,r31,2,0,29
lwzx           r12,r12,r11
mtspr          lr,r12
blrl
```

The compiler rearranges the load from the array, making use of the fact that the address of the array is a constant and can be combined with the -12 (-48 when multiplied by the pointer size) from the index expression. This is a very common transformation that most compilers will do.

Now, the machine code does not show that the compiler wants to load the address of `arr2-48` but it only contains the absolute number which, in this case, happens to be the address of array `arr1`. As a result, StackAnalyzer can not distinguish between `arr1[i]()` and `arr2[i-12]()` and will assume the former alternative, which is clearly wrong.

In this case, StackAnalyzer silently computed incorrect results even for code that was translated from normal C code (that actually appears in practice) by the recommended compilers. Even the symbol information contained in the ELF format can not

solve these problems. To handle this case reliably, range information on the index into the jump table would be required. Obtaining such precise data-flow information from machine code is very hard and impossible in many cases. StackAnalyzer does not seem to do any such analysis. Therefore it can be assumed that with the current state of the art it is not possible to reliably extract call targets from arrays in any but the simplest cases. The new approach presented here handles theses cases without problems.

Conclusion

While the StackAnalyzer is a nice tool and has some advantages in certain cases, the lack of high-level information clearly puts it behind the new approach presented here. Apart from the cases with simple assembly code, vbcc required less or simpler annotations. Also, the annotations needed by vbcc are easy to maintain in the source whereas the separate files required by StackAnalyzer make it difficult to ensure that an annotation stays valid after changes to the application or the build environment.

Also, the diagnostics of vbcc are much better because source level information is available. In all cases, source lines and function names (also of non-external functions) are provided rather than addresses of machine instructions.

When dealing with code produced by tools unknown to StackAnalyzer, many problems can appear, in the worst-case resulting in silent errors that lead to wrong stack sizes. Therefore, the theoretical advantage to analyze code from different tools, assembly code, libraries etc. is not there in practice.

To conclude, the high-level approach presented and implemented in this thesis seems to be superior, although there are also some advantages to the low-level approach. For further improvement a combined approach might be able to combine the best of both worlds. For example, a tool similar to StackAnalyzer could analyze the stack usage of inline assembly and pass those results to the compiler. As all the “trickier” calculations can be done by the compiler, the low-level tool would not have to handle complex cases or recognize code-patterns. Therefore, it would probably be possible to make it safer as well as smaller.

Chapter 5

Context Optimization

The previous chapters have addressed reduction of RAM usage by means of common compiler optimizations as well as static analysis of stack usage. Although very well suited to systems using static operating systems, these methods are of a more general nature and also applicable to other small systems.

This chapter will now describe a new optimization method that has been developed specifically for static operating systems and was first published in [17].

5.1 Task Contexts

While it is possible to use only very few bytes of RAM for most operating system resources, it turns out that the memory needed to store task contexts often makes up the biggest part of RAM usage by the operating system.

When a task is preempted and the operating system switches to another task, it has to store all information needed to switch back to the preempted task later on. Classically this means to store the entire register set of the processor which is available to applications. Interrupt service routines can be viewed similarly as tasks in this respect.

The following microcontrollers used in volume production in cars illustrate that the task contexts can use a significant amount of available RAM (consider that 20 – 30 tasks and interrupt service routines are common figures):

- Motorola MPC555 [105]
  26KB RAM
  32 general-purpose-registers (32bit)
  32 floating-point-registers (64bit)
  ⇒ task context about 384 bytes (1.44% of total RAM)

- Infineon C164CI [75]
  2KB RAM
  16 general-purpose-registers (16bit)
  ⇒ task context about 32 bytes (1.65% of total RAM)

As mentioned above, these chips are actually used in high-volume production with static operating systems. It must be said that there are also microcontrollers using an accumulator architecture offering only very few registers where task contexts are far less critical:

- Motorola 68HC912DG128 [105]
  8KB RAM
  1 accumulator (16bit)
  2 index registers (16bit)
  stack pointer, program counter, flags, ...
  ⇒ task context about 12 bytes (0.15% of total RAM)

However, this architecture is based on an old instruction set. In new designs, even 8bit architectures are starting to use large register sets in order to better suit compiler-generated code:

- Atmel AVR [9]
  128 bytes – 4KB RAM
  32 general-purpose-register (8bit)
  ⇒ task context about 32 bytes (0.8%–25% of total RAM)

It can be observed that new architectures (even smallest ones) tend to have many registers. This trend will most likely continue (see [65]). A test with several embedded applications in [148] suggests that in most cases, only 7 – 19 registers are sufficient even for RISC processors. Therefore, when storing a large register set, it is not unlikely to save registers which do not (and often can not) contain a live value or are not modified by the preempting tasks.

Many embedded systems have a relatively large number of event-triggered real-time tasks or interrupt service routines which execute only very small pieces of code (e.g. fetching a value from a sensor). Compiler optimizations which use a lot of registers (inlining, unrolling, software-pipelining) are rarely used as they often increase code size. Therefore, a large part of the tasks may use only a small part of the register set.

As a result, many systems actually waste valuable RAM for task contexts which could be optimized. Imagine a CPU with 1000 registers: Using conventional techniques, it would need a lot of extra RAM for task-contexts although it might, in fact, never be necessary to store any registers at all during a context-switch.

5.2 Current Practice and Related Work

As the size of task-contexts actually matters in practice (a few bytes of RAM may cost millions of dollars in high volume production), there are already a few attempts to address this issue.

There have been several approaches to improve context-switch times either by software or hardware. However, they are tailored to much bigger systems and do concentrate on speed rather than RAM usage (see e.g. [145, 13, 110, 66]).

Some operating systems allow specification of reduced register sets for some tasks. A common variant is an attribute to specify that a task does not use floating-point registers. However, this solution is neither very fine-grained nor very safe without compiler support. A developer will usually assume that a task will not use floating point registers as long as the source code does not contain any floating point variables or constants.

There are, however, many cases where compilers will use floating point registers for other purposes like in this example:

```
struct coordinate {int x, y;} p1, p2;
...
p1 = p2;
```

On some architectures it is beneficial to use a floating point register to copy such a structure. Therefore, specifying reduced register sets for tasks is only viable if the compiler used actually documents how it handles the registers in question.

Another frequently used technique is to save smaller task contexts when a task voluntarily releases the processor by calling some operating system function. Most ABIs prescribe that any called function may destroy some register values (the "caller-save" registers). Therefore, the compiler will never keep live values in those registers at the point of releasing the CPU. As a result, the operating system has to store only the other part of the register-set, the "callee-save" registers.

This technique, however, can only save some RAM for non-preemptive tasks. While the operating system only needs to save a part of the context, the compiler will save the caller-save registers containing live values on the stack. Execution speed of such non-preemptive context switches is improved by register analysis in [66].

For preemptive tasks, which can also be interrupted at arbitrary times (when all registers might contain live values), it does not reduce RAM usage at all. The space for a full context must still be reserved for that case. Actually, there is some overhead if the task is interrupted just before calling an operating system function. At this point, some of the caller-save registers have already been saved on the task stack by the compiler. Nevertheless, the operating system has to store the entire task-context because the preemption was not caused by calling the operating system. Therefore, some of the caller-save registers are stored twice.

5.3 Context Optimization

The new idea presented here reduces the RAM needed for task contexts through interaction between the compiler and the static operating system. As the system is static and all tasks are known at generation time, it is possible to save a different register set for each task.

When a task is preempted, only registers which contain live values *and* can be destroyed (by preempting tasks) have to be saved. With knowledge about the scheduling mechanism, the API of the operating system etc., the compiler can determine safe bounds for register sets that have to be stored for each task. Furthermore, modified register-allocation may result in smaller task-contexts without sacrificing intra-task code quality.

Using this information for operating system generation allows the allocation of only as much RAM as is needed by the minimized task-contexts. Obviously, the operating system code which performs context-switches will have to be generated accordingly to save/restore different register-sets depending on the preempted task.

For this purposes an embedded system using a static operating system can be modelled using the set of tasks

$$T = \{t_1, ..., t_n\},$$

the register set

$$R = \{r_1, ..., r_m\}$$

and a set of code blocks

$$L = \{l_1, ..., l_k\}$$

which will be illustrated by the following example.

5.3.1 Example

Consider the following (admittedly very small) example of a system with three tasks. Assume fixed priority fully preemptive scheduling with the task priorities given in the code. This implies that task `t1` can be interrupted by tasks `t2` and `t3`, task `t2` can be preempted only by task `t3` and task `t3` is the highest priority task which can never be interrupted.

To illustrate some of the situations that can arise, task `t2` contains a critical section (for example obtaining a mutex using some kind of priority ceiling or priority inheritance protocol) which prevents it from being preempted by task `t3` inside this critical section. Also, tasks `t2` and `t3` share some code, namely the function `f`.

Other situations which could lead to different combinations of preemptions would be tasks entering a blocking state (for example, waiting for a semaphore). All these situations can be formalized using an interference graph which will be described below.

The `alloc` and `free` comments shall indicate the beginning and end of register live ranges. `l1` – `l5` are placeholders for blocks of code which do not change preemptability. These blocks do not have to be straight-line basic blocks but can be rather arbitrary pieces of code as long as the preemptability does not change inside. Of course, a conservative estimate could be used for an entire task, i.e. a context-insensitive approach. However, this would negatively affect the benefits of the optimization. Apparently, this partitioning into code blocks depends on the scheduling mechanism and system services provided by the operating system.

`r1` – `r8` designate the register set. For example, `r7` and `r8` could be floating-point registers (which would explain why they are used rather than `r2` and `r3`) in task `t1`.

```
TASK(t1) /* prio=1 */
{
  /* alloc r1, r7, r8 */
  l1
  /* free r1, r7, r8 */
}
TASK(t2) /* prio=2 */
{
  /* alloc r1 */
  l2
  EnterCriticalSection();
    /* alloc r2, r3 */
    l3
    /* free r2, r3 */
  LeaveCriticalSection();
  f();
  /* free r1 */
}
TASK(t3) /* prio=3 */
{
  /* alloc r1, r2, r3 */
  l4
  f();
  /* free r1, r2, r3 */
}
void f()
{
  /* alloc r4 */
  l5  /* free r4 */
}
```

Figure 5.1 illustrates the situation. There are three columns, one for each task. In each column there are all the code blocks `l1` – `l5` which are executed in this task, together with all registers that are live within each block. Note that the live registers are task-specific for shared code. There are different registers live in `l5` (i.e. the function `f`) because a different set of registers is live at the different call sites of `f`.

As a result, the set of used (or live) registers at each block (as it will be calculated by the compiler) is a mapping from a pair consisting of a task and a code block to a set of registers:

$$U : T \times L \to \mathcal{P}(R),$$
$$(t, l) \mapsto \{r \in R : r \text{ is live in block } l \text{ in task } t\}$$

Additionally, there are edges from every code block in a column to all the tasks which can preempt the task within this block. For example, task `t2` can be preempted in blocks `l2` and `l5` by task `t3` but not in block `l3` due to the critical section.

This interference graph is a mapping from a pair consisting of a task and a code block to a set of tasks:

$$I : T \times L \to \mathcal{P}(T),$$
$$(t, l) \mapsto \{t' \in T : t' \text{ can preempt } t \text{ in block } l\}$$

Different scheduling algorithms and operating systems can be modelled that way and will have significant impact on the interference graph.

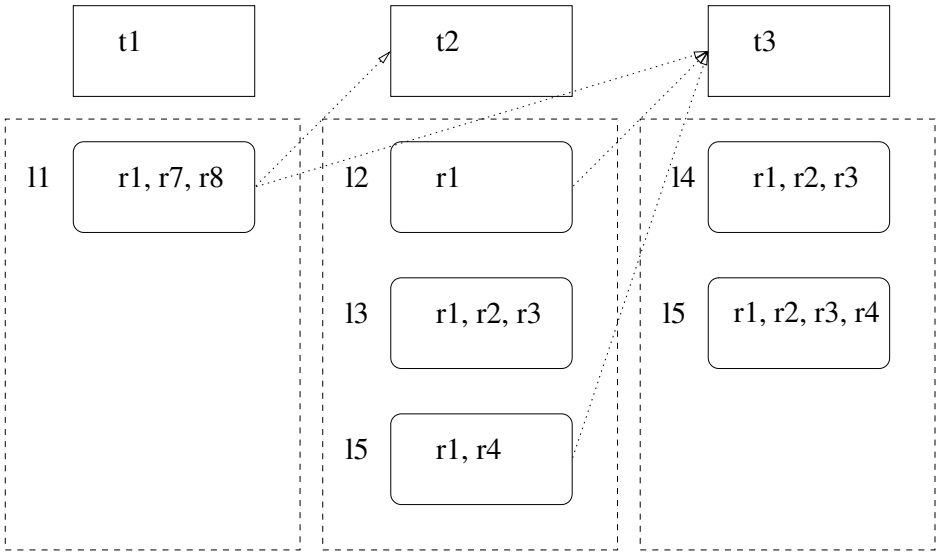


Figure 5.1: Interference Graph

5.3.2 Bounding Task-Contexts

With the model described above, it is possible to specify and calculate an optimized context for each task. First, the set $D(t)$ of all registers each task destroys, is calculated as

$$D(t) := \bigcup_{l \in L} U(t, l).$$

For the small example, one obtains:

$$D(t_1) = \{r1, r7, r8\},$$
$$D(t_2) = \{r1, r2, r3, r4\},$$
$$D(t_3) = \{r1, r2, r3, r4\}$$

When a task t is preempted, it is necessary to store all those registers which are live *and* can be destroyed by any task which can preempt task t. It would be possible to store different register sets depending on the code block where the task was preempted (by looking at the program counter when doing a context-switch). However, this is unlikely to give much benefit and will significantly complicate the context-switch code in the operating system.

Therefore, for each task it is sufficient to traverse all blocks of its code, and add to its context all registers which are live in that block and can be destroyed by any task that can preempt it (i.e. there is a corresponding edge in the interference graph). Formally, the task-context $C(t)$ of a task t (i.e. the set of registers that is sufficient to save whenever task t is preempted) can be written as:

$$C(t) = \{r \in R : \exists l \in L, t' \in I(t, l) : r \in U(t, l) \cap D(t')\}.$$

For the small example, one obtains:

$$C(t_1) = \{r1\},$$
$$C(t_2) = \{r1, r4\},$$
$$C(t_3) = \emptyset$$

Only memory to store three registers is needed. Without this analysis every task-context would need to provide space for the entire register set ($3 \cdot 8$ registers in this example). Obviously the benefit will not always be that big, but in very cost-sensitive systems, a few bytes saved might actually help to fit an application into a smaller chip and reduce costs significantly.

5.3.3 Inter-Task Register-Allocation

So far, the space for task-contexts has been minimized by analyzing the code already produced for the application. The next goal is to further minimize the RAM requirements by already considering the task-contexts when translating the application, especially when assigning registers.

The scope of register-allocation in compilers varies from single expressions or basic blocks to single functions or inter-procedural register-allocation (see e.g. [25, 146, 58, 106]). In this thesis, the scope will be extended to inter-task register-allocation. Similar to inter-procedural assignment of registers which helps to reduce spilling of registers across function-calls, inter-task assignment can help to reduce the memory needed to store registers of preempted tasks.

The goal of this optimization is to minimize the total space of all task-contexts of a system. As the task-contexts of tasks which can not preempt each other (for example tasks with the same priority that do never wait) can use the same memory, the space required to store all task-contexts is not necessarily the sum of the sizes of all contexts.

Let

$$s(r), r \in R$$

be the memory requirement of each register, and

$$\{T_1, ..., T_n\}$$

a partitioning of T, such that all tasks in a partition T_i can not preempt each other, i.e.

$$\forall t \in T_i, l \in L : I(t, l) \cap (T_i \setminus t) = \emptyset.$$

Therefore, if $M(i)$ is the size needed to store the largest task-context in a partition T_i, i.e.

$$M(i) := \max_{t \in T_i} \sum_{r \in C(t)} s(r),$$

then the object of minimization is:

$$\sum_{i=1}^{n} M(i).$$

Inter-task register-allocation should not negatively affect the intra-task code-generation. Typically, it will only guide the choice between otherwise identical registers.

For the small example presented above, a possible improvement would be to replace r_1 by r_5 in t_2 and by r_6 in t_3 (assuming these registers are available). This would minimize the task-contexts to:

$$C(t_1) = \emptyset,$$
$$C(t_2) = \{r_4\},$$
$$C(t_3) = \emptyset$$

Although more registers are used, the total RAM requirements would be reduced.

Unfortunately, this optimization problem is not easily solvable (as it is known, even local register-allocation is usually NP-complete). Therefore, it is necessary to find approximations or solutions for special cases. The scheduling algorithm and system services offered by the operating system may affect inter-task register-allocation in a non-trivial way. A small experimental implementation for one specific scheduling strategy will be described below. The next chapter will present a full implementation for a real commercial OSEK implementation.

5.4 Requirements on Compilers

To carry out these optimizations, a compiler has to be able to calculate good bounds on the registers used in different blocks of a task. The requirements on the compiler are very similar to those needed for the stack analysis that has been presented in the previous chapter. It can only be achieved if a call-tree can be constructed and the registers used are known to the compiler most of the time. Where this is not possible, worst-case assumptions have to be made and good results are hard to obtain.

Applications using static operating systems usually are rather well suited to this kind of static analysis. Neither recursions nor dynamic memory allocations are used due to reasons of safety and efficiency. Also, function pointer variables are generally not used and use of external library functions is very limited (source code is generally available for the entire system).

These restrictions reduce some of the most difficult problems for static analysis. However, there are still a number of requirements on compilers to obtain good results:

- Cross-module analysis is needed as the applications are usually split across files.
- A call-tree has to be built, usually requiring data-flow- and alias-analysis.
- Tasks, the scheduling-mechanism and the operating system services have to be known to enable construction of the interference graph.
- Side-effects (especially register-usage) of inline-assembly (if available), library- and system-functions should be known.

While a few of these features are not yet common in most compilers, more and more modern compilers provide at least the infrastructure (for example, cross-module-optimizations) to incorporate them. Regarding the vbcc compiler, the corresponding sections in the previous chapters apply.

5.5 Experimental Implementation and Results

This section will describe an experimental stand-alone implementation of inter-task register-allocation and minimization of task-contexts in vbcc (MPC555 backend). Some small benchmarks will be used to show the theoretical potential of this optimization. A full implementation for a commercial OSEK system will be presented in the next chapter.

For now, the operating system model shall be a fixed priority fully preemptive scheduler without blocking or dynamic priority changes. The normal intra-task (but inter-procedural) register-allocation of vbcc was extended to use a priority for each register. If a choice between several otherwise identical registers has to be made by the intra-task register-allocator, it will use the register with the highest priority.

The inter-task register-allocation modifies these register priorities for the intra-task allocator. It processes the tasks in priority order and lowers the priorities of registers that have been used by a task. Therefore, subsequently translated tasks will tend to use different registers as long as appropriate unused registers are still available.

While the scheduling model considered here is rather simple, it is possible to extend this mechanism to more complicated schedulers without too much additional effort (a version supporting all OSEK scheduling variants will be shown in the next chapter). Also, many systems are actually using such schedulers, especially if they have to meet hard real-time constraints [24].

To obtain some first benchmarks, different combinations of tasks from the following categories have been created and optimized.

rbuf: A simple task which just fetches a value from an IO port and stores it into a ring-buffer. It uses three general-purpose registers.

mm: Normal floating-point matrix-multiplication. It uses eleven general-purpose registers and three floating-point registers.

int: A task using all general-purpose registers.

all: A task using all general-purpose registers as well as all floating-point-registers.

Classical optimizations like common-subexpression-elimination, loop-invariant code-motion or strength-reduction have been performed. Loop-unrolling has been turned off. Table 5.1 lists the total context sizes with and without optimization. The first four columns show how many tasks of each category are used for a test case. All tasks

Table 5.1: Benchmark results

n_{rbuf}	n_{mm}	n_{int}	n_{all}	RAM_{std}	RAM_{opt}	savings
10	0	0	0	1040	16	98%
0	10	0	0	3360	296	91%
0	0	10	0	1040	936	10%
0	0	0	10	3360	3024	10%
2	2	2	4	2432	1816	25%
4	2	2	2	1968	1168	41%
4	4	2	0	1968	312	84%
6	0	4	0	1040	384	63%
0	6	0	4	3360	1344	60%
3	1	6	0	1272	596	53%
				20840	**9892**	**53%**

have different priorities. The RAM requirements of each task (e.g. stack-space) are not affected by the context-optimization.

The fifth column lists the total task-context size in bytes with conventional allocation. It is assumed that tasks which do not use floating-point are marked accordingly by the application. A smaller context is allocated for these tasks, even without optimization.

Only the general-purpose- and floating-point-registers which are available for the application have been considered. Any special-purpose registers or registers that must not be used by the application are ignored here. As a result, a full context of a task not using floating-point needs 104 bytes and a full context of a task using floating-point needs 336 bytes. Therefore, if n_f denotes the number of tasks using floating-point and n_i the number of tasks using only general-purpose registers, the non-optimized context size can be calculated as:

$$n_f \cdot 336 + n_i \cdot 104.$$

The last column lists the total task-context size using the minimized register sets obtained from the compiler. Inter-task register-allocation was performed, but with this simple scheduling model, it gives additional benefit only in a few cases.

It can be observed that the savings depend a lot on the constellation of tasks. As long as almost every task uses all registers, the benefit will be small. However, with every task that uses only a part of the register set, memory is saved.

The first four rows are rather academic as they use 10 tasks of the same category (including the best and worst case scenarios). However, the remaining rows better reflect typical systems with a number of tasks using the entire register sets as well as some smaller light-weight tasks using only part of the register set.

For several of these constellations, the optimization reduces RAM usage for task-contexts significantly. Tests with a real OSEK implementation are presented in the next chapter.

Chapter 6

Real OSEK Implementation

To demonstrate and verify that the techniques presented in the previous chapters are also viable in practice, they have been used together with a real commercial OSEK implementation. A specially adapted version of the vbcc compiler and a modified version of the 3SOFT ProOSEK operating system was used on the MPC555 microcontroller.

Stack analysis, inter-task register-allocation as well as a few other minor optimizations have been performed in order to reduce the amount of RAM required by systems built with these tools. Also, by inclusion of stack analysis, the user does not need to specify the RAM usage of the tasks.

6.1 ProOSEK

ProOSEK, a commercial OSEK implementation produced and sold by 3SOFT GmbH, was used to demonstrate the feasibility of the techniques presented. It conforms to the current OSEK/VDX specifications OS 2.2, OIL 2.3 and COM 2.2.2. Versions for many different microcontrollers exist. The MPC555 version was used here. ProOSEK is used in the BMW Standard Core, a software architecture used for most ECUs in new BMW cars. Also, DaimlerChrysler as well as other car manufacturers and suppliers use ProOSEK in current products.

A graphical configuration and generation tool is delivered to the user. All standard OSEK attributes as well as some vendor-specific options can be entered with that GUI. These configurations are saved according to the OIL specification and OIL files can also be loaded.

When the configuration is finished, the tool performs consistency checks. These are, however, only performed on the configuration data and therefore only some errors can be found. The source code of the application is not analyzed. If no inconsistencies are detected, a specially optimized kernel code will be generated by the ProOSEK Configurator. Many optimizations are performed to produce a small kernel suitable for deeply embedded systems.

The kernel code is generated as compiler-specific C code, usually containing some compiler-specific extensions or inline-assembly. The kernel code as well as the application code are compiled (usually with the same compiler and compiler options) and linked together to obtain the final ROM image.

The ProOSEK Configurator is delivered as Java bytecode and can therefore be run on many host systems providing a Java virtual machine (see figure 6.1).

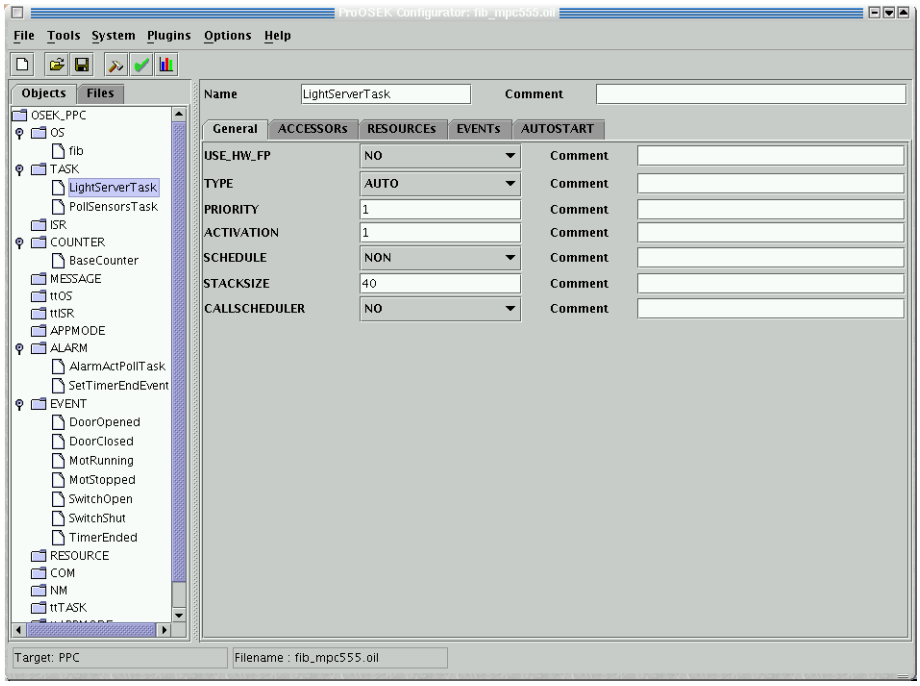


Figure 6.1: ProOSEK Configurator

6.2 Implementation Details

This section describes all the changes that have been made to vbcc and ProOSEK in order to make them work together and minimize the RAM usage of final systems.

6.2.1 Adaption of ProOSEK to vbcc

As was mentioned above, the code generated by ProOSEK contains compiler-specific parts. To support a new compiler, some modifications are usually necessary. Often this includes compiler-extensions, inline-assembly, ABI or assembly-language syntax. The largest part is standard C, however.

The version of ProOSEK for the MPC555 generates code that can be compiled by the DiabData compiler and the GNU compiler (controlled by conditional compilation). Both compilers support the same standard ABI [122] and almost identical assembly-language syntax. Inline assembly is used several times, but no further compiler-specific extensions.

As vbcc supports the same ABI and assembly-language syntax as those compilers, only some inline-assembly had to be adapted. All three compilers use different syntax for inline-assembly. It was, however, simple to provide the necessary alternatives for vbcc.

6.2.2 Stack Analysis

The stack analysis feature that was built into vbcc was used to compute the stack sizes for the OSEK tasks. Therefore, the user does not have to specify a stack size anymore as long as vbcc can determine an upper bound for the stack usage of the task. The values calculated by vbcc are used directly to create stacks of just the size that is needed. See chapter 4 for further details.

All OSEK tasks (apart from endless loops) will call at least one OSEK function that results in a context-switch (`TerminateTask()`) and perhaps many more like `ActivateTask()` or `WaitEvent()`. The stack usage of these functions will also be computed by vbcc (the inline assembly code performing the actual context switch has been annotated).

6.2.3 Task Attributes

Every OSEK task is written in the application code like this:

```
TASK(myTask)
{
  ...
  TerminateTask();
}
```

The `TASK` keyword is implemented by use of the C preprocessor and will expand to some implementation-specific function-definition. This macro has been modified to contain several task attributes that are to be communicated to vbcc.

As a result, when translating a task function, vbcc knows:

- that it is a task,
- its task ID (as used internally by ProOSEK),
- its priority (as used internally by ProOSEK), and
- whether it is preemptive or non-preemptive.

This information is necessary for calculation of task-contexts and stack sizes. Also, vbcc will handle task-functions differently when generating code. Normally, a C function has to store all callee-save registers it uses. For a task, this is not necessary as the task can only terminate, but it will never return to any caller. Therefore, vbcc will not save any registers in a task function.

This is especially interesting because of some oddity of the PowerPC ABI. The machine instruction used for calling a subroutine on the PowerPC does not push a return address on the stack but rather stores it in a special register, the so-called "link-register". If a function calls another function, it has to store this link-register on the stack to preserve its return address. Now, the PowerPC ABI [122] prescribes that this link-register is saved in the stack-frame of the caller (which leaves a stack slot free to be used by the callee for that purpose).

As a result, when starting a task, the stack-pointer usually must not be set to point directly to the top of the stack, but there must be 8 bytes reserved where the task function will store its return address. As vbcc does not store the return address for a task function (an OSEK task never returns, it has to call `TerminateTask()`), wasting these 8 bytes can be eliminated.

6.2.4 Calculation of Task Contexts

Computation of the task contexts is done in a similar fashion to the experimental version described in the previous chapter: The tasks will be translated and analyzed according their priority — starting with the highest priority. All the registers used by the tasks translated so far will be accumulated. Now the context of each task is the intersection of the registers used by this task and all the registers used by all higher priority tasks

so far. The following simple algorithm can be used to compute the minimized task contexts C_i for a fully preemptive system with at most one task per priority. T_{high} is the task with highest priority and T_{low} the task with lowest priority. Register set D is used to keep track of all registers used by higher priority tasks.

$D := \emptyset$
for $T_i := T_{high}, ..., T_{low}$:
  Translate T_i and compute:
    $R :=$ {all registers used by T_i}
  $C_i := R \cap D$
  $D := D \cup R$

However, OSEK provides more complicated scheduling mechanisms. The following facts had to be considered:

- Several tasks can share the same priority.
- OSEK supports preemptive as well as non-preemptive tasks.
- Tasks can wait for events (semaphores) and block.
- The priority of a task can be changed due to the priority ceiling protocol.

Tasks on the same Priority

Handling tasks on the same priority can be done by simply deferring step 5 in the algorithm presented above. Instead, all tasks of the same priority are translated in sequence and another register set P is used to accumulate all registers of all tasks of the current priority. Only after the last task of this priority has been handled, P is added to D ($maxprio$ is the highest priority used by any task):

$D := \emptyset$
for $n := maxprio, ..., 0$:
  $P := \emptyset$
  $\forall T_i, \text{prio}(T_i) = n$:
    Translate T_i and compute:
      $R :=$ {all registers used by T_i}
    $C_i := R \cap D$
    $P := P \cup D$
  $D := D \cup R$

This algorithm takes account for the fact that tasks on the same priority can not interrupt each other (unless one can wait for events which will be discussed below).

Non-preemptive Tasks

Furthermore, OSEK supports non-preemptive tasks. Such tasks will not be interrupted when a task with higher priority gets active. They can, however, call a system function (`Schedule()`) which will allow them to be preempted at this point if a task with higher priority is ready. To handle this case, vbcc will not only compute the total set of registers used by a task, but also the set of all registers that contain live values at a call to `Schedule()`. Only those registers have to be considered for the context of a non-preemptive task. This can yield big improvements in typical code patterns like the following example of co-operative multi-tasking:

```
TASK(T1)
{
  while(1){
    process_1();
    Schedule(); /* allow preemption of higher priority task */
    process_2();
    Schedule(); /* allow preemption of higher priority task */
    process_3();
    Schedule(); /* allow preemption of higher priority task */
  }
}
```

Few or no registers are live across the calls to `Schedule()` in this case. Note however, that all the live registers at the call to `Schedule()` may contain registers that are used further down the call-graph. Consider the following example:

```
TASK(T1)
{
  /* alloc r1, r2 */
  f();
  ...
}

TASK(T2)
{
  /* alloc r1, r2, r3 */
  f();
  ...
}

void f()
{
  /* alloc r1 */
  Schedule();
 ...
}
```

In function `f()` only one register r_1 is live when `Schedule()` is called. However, the context of non-preemptive task `T1` also must contain r_2 as this register was live when `f()`, and therefore also `Schedule()`, was called. For task `T2`, r_1, r_2 and r_3 may be part of the context. This illustrates that local liveness information is not sufficient.

As vbcc translates each function exactly once, this information is passed top-down. i.e. it will first translate `f()` and note that it has r_1 live across a call to `Schedule()`. When `T1` is translated, it is already known that `f()` calls `Schedule()` and the union of r_1 (from `f()`) and the currently live registers in `T1` will be stored as the register set live across a call to `Schedule()` in `T1`.

Waiting Tasks

While non-preemptive tasks and tasks sharing the same priority offer additional opportunities for optimization, i.e. they allow smaller task contexts, tasks waiting for events introduce additional preemptions. Not modelling this mechanism might result

in incorrectly small stack sizes rather than just suboptimal results. A task entering the blocking state (by calling the `WaitEvent()` function) can effectively be preempted by all tasks at this point, even if they have lower priority.

This case is handled very similar to calls to `Schedule()` from non-preemptive tasks. For each task, the set of registers live across a call to `WaitEvent()` is computed. All of those registers that are used by any other task have to be added to the context of this task. Often only few registers will be live across a call to `WaitEvent()` in typical examples like the following one:

```
TASK(T1)
{
  while(1){
    WaitEvent(mySignal);
    /* handle signal */
    ...
  }
}
```

Priority Changes

Calls to `GetResource` or `ReleaseResource` as well as different functions for disabling interrupts can effectively change the priority of a task at run-time. While it would be possible to use this information for further reduction of task contexts, this has not been implemented.

All these mechanisms only restrict possible preemptions. It is only possible to raise the priority of a task over its static priority. Also, this can only be done by the task itself, i.e. when it is already running. Therefore, it can only lock out a higher priority task for some time, but it can not preempt a task higher than its static priority.

To make use of this mechanism, it would be necessary to find the parts of code between, for example, calls to `GetResource` and `ReleaseResource`. This is not a trivial problem and no huge additional improvements are expected. Therefore, it has been omitted and suboptimal results are obtained for such cases at the moment. It may be an option for further research.

Final Algorithm

Combining the different extensions to the original algorithm mentioned in the previous paragraphs, one obtains the following algorithm that has been included in vbcc to compute task-contexts for general OSEK applications. The registers live across calls to `Schedule()` and `WaitEvent()` are computed as S_i and W_i when translating task i. For a non-preemptive task, only registers that are contained in S_i will be added to its context. After all tasks have been translated and therefore all registers used by all tasks are known, registers live across calls to `WaitEvent()` will be added to the corresponding task's context if they are used by any other task. This is necessary because a task could be preempted by every other task when it calls `WaitEvent()`.

$D := \emptyset$
for $n := maxprio, ..., 0$
  $P := \emptyset$
  $\forall T_i, \text{prio}(T_i) = n :$
    translate T_i and compute:
      $R_i := \{$all registers used by $T_i\}$
      $S_i := \{$all registers live across `Schedule()`$\}$
      $W_i := \{$all registers live across `WaitEvent()`$\}$
    if T_i non-preemptive, set
      $C_i := S_i \cap D$
    else
      $C_i := R_i \cap D$
    $P := P \cup R_i$
  $D := D \cup P$

$$\forall i : C_i := C_i \cup \left\{ W_i \cap \bigcup_{j \neq i} R_j \right\}$$

6.2.5 Inter-Task Register-Allocation

To avoid explosion of turn-around times, only simple inter-task register-allocation is performed, much like in the stand-alone implementation (see section 5.3.3). Before translating a task, a higher priority is assigned to registers that already have been used by tasks with higher priority. Therefore, tasks on different priorities (which can usually preempt each other) preferably get assigned different registers while tasks on the same priority tend to share the same registers.

While this is a decent heuristic, it does not accurately model all types of task interference graphs that can occur in OSEK systems. Non-preemptive and waiting tasks are not handled differently. Therefore, it is possible to construct cases where these heuristics yield suboptimal results. There are still opportunities for small improvements. Nevertheless, for many typical cases the results are very good.

6.2.6 Generation of Stacks

Most versions of ProOSEK store task contexts directly on the task stack when a task is interrupted. This is a little bit more efficient than storing the contexts in separate context-save-areas. Also, it has the advantage that sharing of task-contexts is done automatically with sharing of task stacks. Tasks that can not preempt each other can share their stacks as well as memory for their context.

For example, tasks on the same priority that do not call `WaitEvent()` can not preempt each other. Similarly, non-preemptive tasks that do not call `Schedule()` or `WaitEvent()` can share their stacks and contexts. The former case can be detected from the configuration information contained in the OIL file. The task priorities as well as assignment of events to tasks must be statically configured there (the case of a task that has been assigned an event but does never wait for it, can be ignored). The latter case can not be identifier from standard OIL attributes. Some OSEK implementations provide additional proprietary attributes to specify such cases of non-preemptability. They are, however, dangerous to use as it must be ensured that these attributes always correctly reflect the current source code.

During system generation, the ProOSEK Configurator will calculate the stack size for each task by adding the user stack size that is specified in the OIL file, the size of a

task context and additional space that is needed for calling operating system functions. An interference graph, based upon the information available in the OIL file, will be created and optimized stacks are allocated. Where possible, tasks will share the same stack space.

For the modified version, the entire calculation and generation of task stacks was removed from the ProOSEK Configurator and is now handled inside the compiler. The stack size needed by each task is computed using stack analysis and the size of the minimized task contexts is added. No further space for calling operating system functions is needed on the task stack (see below). An optimized stack allocation is then calculated based on these stack sizes and the task information. It is basically identical to the mechanism used in the original ProOSEK Configurator with the exception that vbcc gets the information whether tasks call `Schedule()` or `WaitEvent()` directly from the source code rather than from OIL attributes.

Therefore, user stack calculation, context minimization as well as determination of optimal stack-sharing are done together in the compiler. The data structures needed by ProOSEK will be emitted directly from the compiler during compilation of the system.

6.2.7 Context-Switch Code

When performing a context-switch, the original ProOSEK will basically always store the same register set. There are a few variants as mentioned in section 5.2, but it does not store a different context for every task. However, the compiler has allocated only space for the smaller contexts now. Therefore, it is necessary to store for each task only those registers that are contained in its minimized context.

The compiler will generate small subroutines for every task that load and store only the corresponding registers. Also, a table of function pointers will be emitted. The context-switch code of ProOSEK was completely rewritten to call the right routines for saving/loading registers, based on the ID of the task that is preempted or continued.

Obviously, this increases code size somewhat, but the main objective of this thesis is to reduce RAM requirements. This is reasonable as on-chip RAM is significantly more expensive than ROM. Also, the additional code-size could be reduced by using the following techniques:

- For a set of tasks with similar contexts, it is possible to use the union of these contexts for all tasks in the set and share the routines for saving/restoring the context. This enables fine-tuning between RAM and ROM requirements. As RAM and ROM sizes of a certain microcontroller are fixed, the ability to perform this tuning can be very useful when fitting an application to a certain chip.

- If task-contexts are subsets of another one, it may be possible to use the same routines, just with different entry-points.

- Some architectures (e.g. ARM [8] or 68k [105]) have instructions which can save/restore arbitrary register sets controlled by a bit-mask in the instruction code. In such cases, the ROM overhead can be very small. The PowerPC offers instructions that load or store all general purpose registers from n to 31. If possible, these instructions are used in the context saving routines generated by vbcc. However, no sophisticated attempts have been made to further reduce the code size of these routines because the main focus of this thesis lies in reducing RAM size.

6.2.8 System Calls and System Stack

So far it has been assumed that when a task is preempted, its context gets magically saved and another task is dispatched. Of course, in practice the context-switch is handled by some operating system code. This code, however, will use some stack space and registers itself. This has to be considered when really implementing context minimization.

There are some OSEK system calls that can not lead to rescheduling and are very small. Those calls are usually inlined and use only very few registers and stack. They can be handled simply as part of the task and viewed as application code for these purposes. There are, however, also system calls that are more heavyweight and can lead to a rescheduling.

To keep the RAM usage as small as possible, those ProOSEK system calls have been attributed so that they can be handled specially by the compiler. When such a function is called, vbcc will emit code to store the calling task's context and switch to a special stack for the operating system. As these system functions are executed with interrupts disabled, this stack only has to exist once, no additional space has to be added to the task stacks. Consequently, these system functions are not included in the stack analysis of the tasks (they are assumed to require no stack at all).

One last point to consider is the register usage of those system functions. Although the current task's context is saved before entering such a system function, there is still one problem left. When calculating the minimized context for each task, only registers are included that can be destroyed by other tasks. However, some register might be used by only one task and a system function. In that case the register would not be saved in the task context, but nevertheless overwritten when that system function is called.

To solve that problem, all registers that are used by any system function (that is called by any task or ISR) as well as by a task are added to this task's context. Each register used by system functions will usually increase at most one context. If it is used, for example, by several tasks of different priority (that can preempt each other), the register is already contained in all contexts but the one of the highest priority task. If, on the other hand, the tasks can not preempt each other, there stacks will usually be shared anyway (there are some theoretical exceptions because there can be more than one possibility to share the stacks).

6.2.9 Interrupts

Apart from tasks, OSEK also provides interrupt service routines that are usually triggered by some hardware interrupt request. For the purposes of this thesis they can be handled just like tasks. Comparable changes have been made to the handling of interrupt service routines in ProOSEK.

6.3 Results

Some tests have been carried out to measure the savings that are possible in the real OSEK implementation. For the examples that do not contain ISRs, the correct execution of the applications produced with this system has been tested on a chip simulator. The examples requiring interrupts have been run on an evaluation board with a MPC555 microcontroller.

Unfortunately real OSEK applications have not been available for testing. OSEK has been in production use for only a few years now and the first OSEK applications that

were built into cars are still in production. Obviously, car manufacturers and suppliers do not publish their intellectual property. Therefore, only synthetic benchmarks and examples that are used for demonstration and teaching purposes have been available for tests at this point in time.

The RAM requirement for stacks and optimized contexts has been compared to the requirements of the unmodified ProOSEK. The standard ProOSEK needs stack usage information for every task specified in the OIL file. For the following tests, the real stack size computed by vbcc has been used here because there was no information available on how precise a programmer would have been able to calculate the stack usage. Therefore, no savings due to precise analysis of task stack usage are included in these tests. The savings that are obtained in these tests are due to the following reasons:

- The compiler translates the top level task functions differently (no unneeded saving of registers etc.).
- Some system calls are translated differently to switch to a different stack. Also, they do not save unneeded registers at top level.
- The stack needed for the system calls is computed exactly and only for those system calls that are actually used.
- Reduced task contexts are computed and used for every task. This is the most important factor to reduce RAM usage.

6.3.1 Task Constellations

The previous chapter contains a table of different task combinations that were used to illustrate the possible savings when using minimized task contexts depending on the percentage of lightweight and heavyweight tasks. The theoretical results showed huge saving if there are many lightweight tasks and still significant savings if there are some lightweight tasks in the system (it should be noted that, for example, ISRs often fall into the lightweight category). The results give an estimate of the savings that can be obtained depending on the kind of application, but they assume a theoretical implementation without any operating system overhead. In practice, however, tasks will call operating system functions that will use registers and stack. Therefore, it has to be tested how much of the theoretical savings are retained in a real implementation.

To get these results, the same task combinations as in table 5.1 have been converted to real OSEK tasks. Each task was set to fully preemptive and was assigned a unique priority. For the unmodified ProOSEK, the exact stack size was specified for each task. Furthermore, calls to `ActivateTask()` and `TerminateTask()` have been added to each task. Table 6.1 shows the results. Of course, the RAM usage is higher than in the theoretical results. Also, the savings are somewhat smaller but still significant in all cases. The savings in the theoretical best case scenarios consisting only of lightweight tasks are a bit smaller due to the overhead of the operating system. More important, however, are the savings for the scenarios with some lightweight and some heavyweight tasks. Due to the other optimizations, the savings for those more realistic constellations are even bigger than in the theoretical case without an operating system.

6.3.2 LED Example

This test case is delivered as an example application with ProOSEK. It controls a light running from left to right and back again between positions that can be adjusted by

Table 6.1: Benchmark results revisited

n_{rbuf}	n_{mm}	n_{int}	n_{all}	RAM_{std}	RAM_{opt}	savings
10	0	0	0	2120	452	79%
0	10	0	0	4760	984	79%
0	0	10	0	2120	1396	34%
0	0	0	10	4760	3700	22%
2	2	2	4	3704	1900	49%
4	2	2	2	3176	1188	63%
4	4	2	0	3176	812	74%
6	0	4	0	2120	796	62%
0	6	0	4	4760	1972	59%
3	1	6	0	2384	1032	57%
				33080	**14232**	**57%**

Table 6.2: LED Example Results

	std. impl.	opt. impl.	savings
stack and context	328	208	37%
total RAM	400	280	30%

DIP switches on an evaluation board. This example is written to make best use of ProOSEK features to create an executable requiring only a small amount of RAM. An ISR is triggered by a timer interrupt and will activate periodic tasks to check the switches and set the LEDs. Both tasks are configured as non-preemptive. Additionally, a special ProOSEK attribute is set that tells the system generator that the tasks do not call the OSEK function `Schedule()`. This allows the system generator to share the stacks and contexts of those tasks. Furthermore, non-preemptive tasks need a smaller stack in the default implementation of ProOSEK as caller-save registers do not have to be saved. Table 6.2 shows the size needed for stack and contexts in the original ProOSEK compared to the optimized implementation. The total RAM size of the entire system (i.e. including the static data structures of the operating system) is also included to illustrate the importance of the stack and context size.

6.3.3 Interior Lights

This test case simulates an ECU controlling the interior lights of a car on an evaluation board. It is used for OSEK training courses. The lights are simulated by LEDs on the board, and DIP switches are used to simulate door sensors and light switches. Like the previous example, it uses non-preemptive tasks and an ISR. Additionally, this test case uses several OSEK events and an extended task (which precludes stack sharing). Table 6.3 shows the results for this test case. Significant savings can be obtained because vbcc detects for example that only few registers are live across a call to `WaitEvent()`.

Table 6.3: Interior Lights Results

	std. impl.	opt. impl.	savings
stack and context	484	264	45%
total RAM	570	350	39%

Chapter 7

Conclusion and Future Work

This thesis has presented several mechanisms to reduce the RAM usage of systems using static operating systems by the use of advanced compilation techniques. Common compiler optimizations have been examined regarding their impact on RAM consumption. A high-level stack analysis has been implemented that easily obtains safe and precise bounds for stack usage of application tasks. Safety margins can be eliminated when results of static analysis are available. New optimization algorithms to reduce the RAM needed to store contexts of preempted tasks have been developed and implemented. By eliminating unnecessary storage for unused registers, task contexts can be minimized.

The techniques proposed in this thesis have been implemented in a real production quality compiler and several tests and comparisons have been made. The stack analysis feature was compared with a commercial post link-time tool and showed clear benefits. Optimization of task contexts and inter-task register-allocation can obtain large benefits depending on the number of light-weight tasks in a system.

All techniques have been combined in a real world implementation working with a commercial OSEK system. Tests on real hardware have shown that the algorithms really work and all tricky low-level details can be handled. The new system does not only offer increased usability and maintainability, but it also demonstrates that significant reductions of RAM requirements can be achieved in practice.

There is, however, still room for further improvement. The stack analysis feature could be enhanced by the capability to calculate stack requirements of simple inline-assembly constructs. Only relatively simple heuristics for inter-task register-allocation have been used. More ambitious algorithms could be researched. Support for other static operating systems as well as different OSEK implementations could be implemented. Whether more precise computation of task preemptability, e.g. analyzing critical sections, could produce significant additional savings, is also still an open question. Finally, more tests of complete ECUs would be most desirable. Currently, such applications are universally up-to-date high-end technology that is out of reach for publication, but in some time, real application code might become available. Unfortunately, no reasonable OSEK benchmarks (e.g. like SPECCPU for general compiler optimizations, see [131]) are available.

Bibliography

[1] http://www.3soft.de

[2] http://www.absint.de

[3] http://www.acceleratedtechnology.com

[4] A. V. Aho, R. Sethi, J. D. Ullman. A formal approach to code optimization. *SIGPLAN Notices*, vol. 5, no. 7, pp. 86–100, Jul 1970.

[5] F. E. Allen. Control Flow Analysis. *SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul 1970.

[6] F. E. Allen. Interprocedural data flow analysis. *Information Processing 74*, North-Holland, Amsterdam, pp. 398–402, 1974.

[7] http://www.apple.com

[8] http://www.arm.com

[9] http://www.atmel.com

[10] D. August, M. Vachharajani, N. Vachharajani, S. Triantafyllis. Compiler optimization-space exploration. *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, San Francisco, pp. 204–215, 2003.

[11] M. Auslander, M .Hopkins. An overview of the PL.8 compiler. *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 22–31, 1982.

[12] A. Ayers, S. de Jong, J. Peyton, R. Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 301–312, 1998.

[13] T. Baker, J. Snyder, D. Whalley. Fast Context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, pp. 35–42, Feb 1995.

[14] J. E. Ball. Predicting the effects of optimization on a procedure body. *ACM SIGPLAN Notices*, vol. 14, no. 8, pp. 214–220, 1979.

[15] R. Bannatyne. Microcontrollers for the Automobile. *Micro Control Journal*, 2003. http://www.mcjournal.com/articles/arc105/arc105.htm

[16] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, vol. 21, no. 9, pp. 724–736, 1978.

[17] V. Barthelmann. Inter-task register-allocation for static operating systems. *ACM SIGPLAN Notices*, vol. 37, no. 7, pp. 149–154, 2002.

[18] Supercomputers set new goals. *BBC Go Digital*, Dec 2002. `http://news.bbc.co.uk/2/hi/technology/2565145.stm`

[19] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, R. B. Grove, S. O. Hobbs, W. B. Noyce. The GEM Optimizing Compiler System. *Digital Technical Journal*, vol. 4, no. 4, 1992.

[20] J. Blieberger, R. Lieger. Worst-Case Space and Time Complexity of Recursive Procedures. *Real-Time Systems*, vol. 11, no. 2, pp. 115–144, 1996.

[21] M. Blume, A. W. Appel. Lambda-splitting: a higher-order approach to cross-module optimizations. *ACM SIGPLAN Notices*, vol. 32, no. 8, pp. 112–124, 1997.

[22] `http://www.bound-t.com`

[23] M. M. Brandis. Optimizing Compilers for Structured Programming Languages. *Dissertation*, ETH Zürich, 1995.

[24] L. P. Briand, D. M. Roy. Meeting Deadlines in Hard Real-Time Systems, *IEEE Computer Society Press*, 1999.

[25] P. Briggs. Register Allocation via Graph Coloring. *Dissertation*, Rice University, Houston, 1992.

[26] P. Briggs, K. D. Cooper. Effective partial redundancy elimination. *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 159–170, 1994.

[27] Qiong Cai, Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, San Francisco, pp. 91–102, 2003.

[28] D. Callahan, K. D. Cooper, K. Kennedy, L. Torczon. Interprocedural constant propagation. *ACM SIGPLAN Notices*, vol. 21, no. 7, pp. 152–161, 1986.

[29] P. R. Carini, M. Hind. Flow-sensitive interprocedural constant propagation. *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 23–31, 1995.

[30] P. P. Chang, W.-W. Hwu. Inline function expansion for compiling C programs. *ACM SIGPLAN Notices*, vol. 24, no. 7, pp. 246–257, 1989.

[31] R. Chapman, R. Dewar. Re-engineering a Safety-Critical Application with SPARK95 and GNORT. *Lecture Notes in Computer Science* 1622, Reliable Software Technologies — Ada-Europe '99, Springer-Verlag, pp. 39–51, 1999.

[32] R. Chapman. SPARK and Abstract Interpretation — A White Paper. *Praxis Critical Systems Limited*, Sep 2001.

[33] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. A. Henzinger, J. Palsberg. Stack Size Analysis for Interrupt-driven Programs. *Proceedings of the Tenth International Static Analysis Symposium* (SAS), Lecture Notes in Computer Science 2694, Springer-Verlag, 2003, pp. 109-126.

[34] G. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Symposium on Compiler Construction*, pp. 98–105, Boston, Jun 1982.

[35] L. N. Chakrapani, P. Korkmaz, V. J. Mooney, III, K. V. Palem, K. Puttaswamy, W. F. Wong. The emerging power crisis in embedded processors: what can a poor compiler do? *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, Atlanta, pp. 176–180, 2001.

[36] J. Cho, Y., D. Whalley. Efficient register and memory assignment for non-orthogonal architectures via graph coloring and MST algorithms. *ACM SIGPLAN Notices*, vol. 37, no. 7, pp. 130–138, 2002.

[37] F. C. Chow. Minimizing register usage penalty at procedure calls. *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, Atlanta, pp. 85–94, 1988.

[38] F. C. Chow, J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 501–536, Oct 1990.

[39] J. Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 20–24, 1970.

[40] J. Cocke, K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, vol. 20, no. 11, pp. 850–856, 1977.

[41] A. J. Cockx. Whole program compilation for embedded software: the ADSL experiment. *Proceedings of the ninth international symposium on Hardware/software codesign*, Copenhagen, pp. 214–218, 2001.

[42] `http://www.compilers.de/vbcc`

[43] K. D. Cooper, K. Kennedy. Fast interprocedural alias analysis. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Austin, pp. 49–59, 1989.

[44] K. D. Cooper, P. J. Schielke, D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. *Workshop on Languages, Compilers, and Tools for Embedded Systems*, Atlanta, pp. 1–9, 1999.

[45] K. D. Cooper, L. T. Simpson, C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 5, pp. 603–625, 2001.

[46] `http://www.cosmic.fr`

[47] `http://www.crossware.com`

[48] Dibyendu Das. Technical correspondence: Function inlining versus function cloning. *ACM SIGPLAN Notices*, vol. 38, no. 6, pp. 23–29, 2003.

[49] J. W. Davidson, S. Jinturkar. An Aggressive Approach to Loop Unrolling. *Technical Report: CS-95-26*, University of Virginia, 1995.

[50] J. J. Dongarra, A. R. Hinds. Unrolling Loops in FORTRAN. *Software — Practice and Experience*, vol. 9, no. 3, pp. 219–226, 1979.

[51] D. J. Eggemont (ed.). Embedded Systems Roadmap 2002. `http://www.stw.nl/progress`.

[52] 8-Bit-Markt weiter sehr lebendig. `http://www.elektronikpraxis.de`

[53] 32-Bit-MCUs legen kräftig zu. `http://www.elektronikpraxis.de`

[54] `http://www.elena-fractals.it`

[55] Embedded C++ Technical Committee. The Embedded C++ specification. `http://www.caravan.net/ec2plus`.

[56] J. Esparza, A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Boston, pp. 1–11, 2000.

[57] `http://www.etas.de`

[58] C. Fischer, S. Kurlander. Minimum Cost Interprocedural Register Allocation, *Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 230–241, 1996.

[59] M. Franz. Compiler Optimizations Should Pay for Themselves — Applying the Spirit of Oberon to Code Optimization by Compiler. *Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference*, Universitätsverlag Ulm, pp. 111–121, 1994.

[60] C. Fraser, D. Hanson. lcc, A Retargetable Compiler for ANSI C. `http://www.cs.princeton.edu/software/lcc`.

[61] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, vol. 17, no. 11, pp. 638–642, Nov 1974.

[62] P. B. Gibbons, S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *ACM SIGPLAN Notices*, vol. 21, no. 7, pp. 11–16, 1986.

[63] D. W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 122–133, 1997.

[64] `http://www.google.de`

[65] D. Greene, T. Mudge, M. Postiff. The Need for Large Register Files in Integer Codes, *Technical Report CSE-TR-434-00*, University of Michigan, 2000.

[66] D. Grunwald, R. Neves. Whole-Program Optimization for Time and Space Efficient Threads. *Architectural Support for Programming Languages and Operating Systems*, Cambridge, Massachusetts, pp. 50–59, 1996.

[67] W. Harrison. A new strategy for code generation: the general purpose optimizing compiler. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, Los Angeles, pp. 29–37, 1977.

[68] H. Heinecke, A. Schedl, J. Berwanger, M. Peller, V. Nieten, R. Belschner, B. Hedenetz, P. Lohrmann, C. Bracklo. FlexRay — ein Kommunikationssystem für das Automobil der Zukunft. *Elektronik Automotive*, Sep 2002.

[69] J. L. Hennessy, T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, pp. 422–448, 1983.

[70] A. M. Holler. Compiler optimizations for the PA-8000. *COMPCON Spring 97*, San Jose, pp. 87–94, 1997.

[71] L. P. Horwitz, R. M. Karp, R. E. Miller, S. Winograd. Index Register Allocation. *Journal of the ACM*, vol. 13, no. 1, 1966.

[72] J. Hughes, L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. *International Conference on Functional Programming*, Paris, France, pp. 70–81, 1999.

[73] A. Humphreys. Registervergabe durch Graphfärbung für verschiedene Mikrocontroller. *Studienarbeit, Uni-Erlangen*, to appear.

[74] `http://www.ibm.com`

[75] `http://www.infineon.com`

[76] `http://www.intel.com`

[77] ISO/IEC 8652:1995. Programming Languages — Ada.

[78] ISO/IEC DTR 15942. Programming Languages — Guide for the Use of Ada.

[79] ISO/IEC 9899:1999. Programming Languages — C.

[80] ISO/IEC. Rationale for International Standard — Programming Languages — C, Revision 2, Oct 1999.

[81] M. Jones. What really happened on Mars Rover Pathfinder. *The Risks Digest*, vol. 19, no. 49, 1997.

[82] D. Kästner,. S. Wilhelm. Generic control flow reconstruction from assembly code. *ACM SIGPLAN Notices*, Vol. 37 , no. 7, Jul 2002.

[83] G. A. Kildall. A unified approach to global program optimization. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, Boston, pp. 194–206, 1973.

[84] J. Knoop, O. Rüthing, B. Steffen. Lazy code motion. *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 224–234, 1992.

[85] J. Knoop, O. Rüthing, B. Steffen. Partial Dead Code Elimination. *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, pp. 147–158, 1994.

[86] Kraftfahrt-Bundesamt. Pressebericht 2003/2004. *Pressestelle Kraftfahrt-Bundesamt*, Flensburg, 2003.

[87] M. Kreuzer. Entwicklung des Embedded-Software-Marktes — Vertikale Software-Pakete im Trend. *Markt & Technik*, pp. 32–33, vol. 50, 2003.

[88] S. R. Ladd. An Evolutionary Analysis of GNU C Optimizations. `http://www.coyotegulch.com/acovea/index.html`, Nov 2003.

[89] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, Atlanta, pp. 318–328, 1988.

[90] Chingren Lee, Jenq Kuen Lee, Tingting Hwang, Shi-Chun Tsai. Compiler optimization on VLIW instruction scheduling for low power. *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 2, pp. 252–268, 2003.

[91] R. Leupers, P. Marwedel. Function inlining under code size constraints for embedded processors. *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, San Jose, pp. 253–256, 1999.

[92] R. Leupers. Code Optimization Techniques for Embedded Processors — Methods, Algorithms, and Tools. *Kluwer*, 2000.

[93] R. Leupers. LANCE: A C Compiler Platform for Embedded Processors. *Embedded Systems/Embedded Intelligence*, Nürnberg, Feb 2001.

[94] D. Liang, M. J. Harrold. Efficient points-to analysis for whole-program analysis. *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 199–215, 1999.

[95] `http://www.livedevices.com`

[96] V. B. Livshits, M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 317–326, 2003.

[97] E. S. Lowry, C. W. Medlock. Object code optimization. *Communications of the ACM*, vol. 12, no. 1, pp. 13–22, Jan 1969.

[98] F. Luccio. A comment on index register allocation. *Communications of the ACM*, vol. 10, no. 9, 1967.

[99] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W.-M. W. Hwu. Compiler code transformations for superscalar-based high performance systems. *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Minneapolis, pp. 808–817, 1992.

[100] S. McFarling, J. Hennessy. Reducing the cost of branches. *ACM SIGARCH Computer Architecture News*, vol. 14, no. 2, pp. 396–403, 1986.

[101] S. McFarling. Program optimization for instruction caches. *ACM SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 183–191, 1989.

[102] `http://www.metrowerks.com`

[103] Guidelines For The Use Of The C Language In Vehicle Based Software. *The Motor Industry Software Reliability Association*, Apr 1998.

[104] E. Morel, C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, vol. 22, no. 2, pp. 96–103, 1979.

[105] `http://www.motorola.com`

[106] S. Muchnick. Advanced compiler design and implementation, *Morgan Kaufmann Publishers*, 1997.

[107] E. M. Myers. A precise inter-procedural data flow algorithm. *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Williamsburg, pp. 219–230, 1981.

[108] J. Nievergelt. On the automatic simplification of computer programs. *Communications of the ACM*, vol. 8, no. 6, pp. 366–370, Jun 1965.

[109] `http://www.nullstone.com/eqntott/eqntott.htm`

[110] P. Nuth. The Named-State Register File, *thesis*, MIT, 1993.

[111] OPTIMIST Project. `http://www.ida.liu.se/~chrke/ceniit`

[112] The OOPS! site. `http://oops.tepkom.ru/index.html`

[113] `http://www.osek-vdx.org`

[114] OSEK/VDX. Operating System Specification, Version 2.2.1, Jan 2003.

[115] OSEK/VDX. Time-Triggered Operating System Specification, Version 1.0, Jul 2001.

[116] OSEK/VDX. OIL: OSEK Implementation Language, Version 2.4.1, Jan 2003.

[117] OSEK/VDX. Network Management, Concept and Application Programming Interface, Version 2.5.2, Jan 2003.

[118] OSEK/VDX. Communication Specification, Version 3.0.2, Dec 2003.

[119] E. Özer, A. P. Nisbet, D. Gregg. Classification of Compiler Optimizations for High Performance, Small Area and Low Power in FPGAs. *Technical Report*, Department of Computer Science, Trinity College, Dublin, Ireland, Jun 2003.

[120] J. Palsberg. Resource-Aware Compilation. `http://www.cs.ucla.edu/~palsberg/resource-aware-compilation.html`

[121] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149–206, 2001.

[122] PowerPC Embedded Application Binary Interface. Motorola, Oct 1995.

[123] T. Proebsting. Proebsting's Law. `http://research.microsoft.com/~toddpro/`

[124] J. Regehr, A. Reid, K. Webb. Eliminating stack overflow by abstract interpretation. *Proceedings of the 3rd International Conference on Embedded Software*, Philadelphia, pp. 306–322, Oct 2003.

[125] `http://www.renesas.com`

[126] V. Sarkar. Optimized unrolling of nested loops. *Proceedings of the 14th international conference on Supercomputing*, Santa Fe, pp. 153–166, 2000.

[127] B. Scholz, E. Eckstein. Register allocation for irregular architectures. *ACM SIGPLAN Notices*, vol. 37, no. 7, pp. 139–148, 2002.

[128] `http://www.sgi.com`

[129] D. Shaporenkov. Interprocedural Optimizations in PROMISE Framework. *Master Thesis*, St. Petersburg State University, 2002 (in Russian).

[130] The SPAM Project. `http://www.ee.princeton.edu/spam`

[131] `http://www.specbench.org`

[132] `http://www.specbench.org/osg/cpu95/news/eqntott.html`

[133] O. Spinczyk, W. Schröder-Preikschat, D. Beuche, H. Papajewski. PURE/OSEK - Eine aspektorientierte Betriebssystemfamilie für Kraftfahrzeuge. *Proceedings of the GI Workshop on "Automotive SW Engineering & Concepts"*, Frankfurt/Main, in INFORMATIK 2003 (GI Lecture Note in Informatics), ISBN 3-88579-363-6, pp. 330–334, Sep 2003.

[134] F. Stappert, P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *C-LAB Report*, 27/97, Dec 1997.

[135] S. Steinke, L. Wehmeyer, P. Marwedel. Energieeinsparung durch neue Compiler-Optimierungen. *Elektronik*, 13/2001, pp. 62–67.

[136] G. Stelzer. Allgegenwärtige Rechenzwerge. *Elektronik* 13/2003, pp. 68–71.

[137] G. Stelzer. Leistungsfähige Rechenknechte. *Elektronik*, 18/2003, pp. 66–71.

[138] The SUIF Group. `http://suif.stanford.edu`

[139] S. Tikal. Trend hin zur RISC-Architektur. *Markt & Technik*, 35/2003, p. 8.

[140] TIS Committee. DWARF Debugging Information Format Specification, Version 2.0. 1995.

[141] J.-P. Tremblay, P. G. Sorenson. The Theory and Practice of Compiler Writing. *McGraw-Hill*, 1985.

[142] L. Unnikrishnan, S. D. Stoller, Y. A. Liu. Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages. Indiana University, Computer Science Dept., Technical Report 538, April 2000.

[143] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, F. Catthoor. Advanced copy propagation for arrays. *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 24–33, 2003.

[144] `http://www.vector-informatik.de`

[145] C. Waldspurger, W. Weihl. Register Relocation: Flexible Contexts for Multithreading, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, pp. 120–130, May 1993.

[146] D. Wall. Global Register Allocation at Link Time. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, pp. 264–275, 1986.

[147] M. N. Wegman, F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 2, pp. 181–210, 1991.

[148] L. Wehmeyer, M.K. Jain, S. Steinke, P. Marwedel, M. Balakrishnan. Analysis of the Influence of Register File Size on Energy Consumption, Code Size and Execution Time. *IEEE TCAD*, vol. 20, no. 11, Nov 2001.

[149] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Las Vegas, pp. 83–94, 1980.

[150] B. A. Wichmann. High Integrity Ada. *National Physical Laboratory*, Teddington, Mar 1997.
`http://anubis.dkuug.dk/JTC1/SC22/W49/HRG-High-Integrity-Ada.html`

[151] `http://www.windriver.com`

[152] A. P. Yershov. ALPHA — An Automatic Programming System of High Efficiency. *Journal of the ACM*, vol. 13, no. 1, pp. 17–24, Jan 1966.

[153] Zephyr: Tools for a National Compiler Infrastructure.
`http://www.cs.virginia.edu/zephyr`

[154] W. Zhang, G. Chen, M. Kandemir, M. Karakov. Interprocedural optimizations for improving data cache performance of array-intensive embedded applications. *Proceedings of the 40th conference on Design automation*, Anaheim, pp. 887–892, 2003.

[155] W. Zhang, G. Chen, M. Kandemir, M. Karakov. A compiler approach for reducing data cache energy. *Proceedings of the 17th annual international conference on Supercomputing*, San Francisco, pp. 76–85, 2003.

[156] M. Zhao, B. Childers, M. L. Soffa. Predicting the impact of optimizations for embedded systems. *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 1–11, 2003.

[157] K. M. Zuberi, P. Pillai, K. G. Shin, W. Nagaura, T. Imai, S. Suzuki. EMERALDS-OSEK: A Small Real-Time Operating System for Automotive Control and Monitoring. *SAE Technical Paper Series*, 1999-01-1102, Mar 1999.

Lebenslauf

Persönliche Daten

Name:	Volker Barthelmann
Anschrift:	Heinrich-Soldan-Str. 16B, 91301 Forchheim
Geburtstag:	14.10.1972
Geburtsort:	Forchheim
Staatsangehörigkeit:	deutsch

Beruflicher Werdegang

seit 02/2004:
Selbständiger Berater/Entwickler für eingebettete Systeme

01/2001-01/2004:
Wissenschaftlicher Mitarbeiter
Lehrstuhl für Programmiersprachen, Universität Erlangen-Nürnberg

05/1998-12/2003:
Software-Entwicklungsingenieur
3SOFT GmbH, Erlangen

03/1997-02/1998:
Wissenschaftliche Hilfskraft
Lehrstuhl für angewandte Mathematik, Universität Erlangen-Nürnberg

Ausbildung

01/2001-01/2004:
Doktorand der Informatik, Universität Erlangen-Nürnberg
Abschluss: Doktor-Ingenieur

10/1992-04/1998:
Studium der Mathematik, Universität Erlangen-Nürnberg
Abschluss: Diplom-Mathematiker (univ.)

09/1983-08/1992:
Ehrenbürg Gymnasium Forchheim
Abschluss: Abitur

09/1979-08/1983:
Anna-Grundschule Forchheim