Lund University

Master's Thesis

Emulator Speed-up Using JIT and LLVM

Author
Hans Wennborg

Adviser
Krister Walfridsson
ARM Sweden

Examiner
Dr Jonas Skeppstedt
Lund University

January 2010

Abstract

English

This thesis investigates compiler optimisations applied to a JIT-compiling emulator which is used in a video-coding system. The JIT compiler is built using LLVM which provides some readily available optimisation passes. Some of these are evaluated, some are patched, and some new ones are implemented. The optimisations are evaluated by running a number of benchmarks. By applying optimisations that target redundancies introduced by the JIT-compiler, as well as some optimisations not performed on the firmware originally, a benchmark execution speed-up of between 20 and 70 % is achieved.

Swedish

Detta examensarbete undersöker kompilatoroptimeringar applicerade på en JIT-kompilerande emulator som används i ett system för videokodning. JIT-kompilatorn är byggd med hjälp av LLVM, som erbjuder flera lättillgängliga optimeringspass. Några av dessa utvärderas, några patchas, och några nya implementeras. Optimeringarna utvärderas genom att köra olika benchmarks. Genom att applicera optimeringar som riktar sig mot redundans introducerad av JIT-kompilatorn, samt optimeringar som inte gjorts på firmwaren ursprungligen, uppnås en uppsnabbning av körtiden på mellan 20 och 70 %.

Preface

There are two persons who have been especially influential in the conception of this thesis work: Jonas Skeppstedt and Krister Walfridsson. Jonas Skeppstedt's courses have continuously fueled my affection for beautifully written and well performing software. In particular, his courses on optimising compilers and algorithm implementation have given me the urge to reduce cycle counts. Krister Walfridsson and I first discussed LLVM during a company Christmas party at ARM Sweden (called Logipard at the time) in 2007. Since then, it has come up now and again and when I found out that it was being used in their emulator, and that there was potential for a master's thesis, I was thrilled. Thank you both for the inspiration and support I have received during this project.

I am also very grateful for the support received from the LLVM community through the `#llvm` IRC channel and the LLVM developers' mailing list.

Finally, thanks to ARM Sweden who have been kind enough to have me in the office, tinkering with their software.

This work is dedicated to the IBM Personal System/2 Model 50. Thank you for getting me started!

Lund, Sweden
January 2010

Contents

Chapter 1

Introduction

This master's thesis investigates the possibility of increasing performance using compiler optimisations in an emulator system that uses LLVM to perform JIT-compilation of the emulated code.

Getting a speed-up is valuable to users of the emulator, who are typically software developers or test engineers. Increased emulator performance allows developers and testers to be more effective in their work. As JIT-compilation and LLVM are both popular technologies, it is interesting to see what kind of optimisations are worthwhile in this setting.

This thesis is organised as follows: A background to the technology used in this thesis is given in Chapter 2. Chapter 3 specifies the purpose of the thesis and defines how measurements are made and what benchmarks are used. The system which is the concern of this thesis is described in detail in Chapter 4. The main part of this thesis is Chapter 5 which describes and evaluates optimisations. Finally, conclusions are made in Chapter 6.

Chapter 2

Background

2.1 The Video Engine

ARM Sweden develops a system for encoding and decoding video. This system is used in products such as mobile phones where it allows for playback of video and encoding of video captured using the device's camera. Using special hardware for video coding frees up the CPU of the host system, is much more energy efficient, and in some cases allows video processing that would not be possible using the host's processor.

An emulator is used during the development of the system. This allows for early development and testing of the system's firmware without the need for actual hardware.

Until sometime late 2008, the emulator performed simple interpretation of the firmware code, but today it does JIT-compilation using LLVM to achieve better performance.

2.2 Just-In-Time Compilation (JIT)

The term just-in-time (JIT) has long been used in business context to describe the idea of manufacturing products just in time for when they are needed, thereby reducing inventory costs.

In computer software, JIT refers to the concept of somehow compiling code on demand as a program is running, rather than doing it before-hand. The history of JIT compilation can be traced back to the 1960s [5], but it is today perhaps most widely known for its use in virtual machines used for executing Java programs.

Java programs are compiled to a format called byte code. This byte code is then run by a virtual machine. The virtual machine can either interpret the byte code or, when it sees fit, compile the byte code into native machine code which is then executed. This execution of native code is typically much faster than interpreting the original byte code. The virtual machine must be careful to handle the trade-off between the extra time to compile to native machine code and the benefit of faster execution.

2.3 The Low-Level Virtual Machine (LLVM)

The Low-level Virtual Machine (LLVM) is a framework for optimisation, compilation and execution of programs. It was introduced by Chris Lattner in his master's thesis of 2002 [7]. As its name suggests, LLVM can be used as a virtual machine to execute programs represented in a virtual instruction set, in this case LLVM IR (for intermediate representation). Such programs can be executed by LLVM using interpretation or JIT-compilation. It can also compile the programs off-line for later execution.

One use for LLVM is as a compiler back-end. For example, Clang [1] is a front-end for C, C++ (parts of it) and Objective C, that produces LLVM IR, which is then optimised and compiled to native code. Another front-end is llvm-gcc, which is the GNU Compiler Collection (GCC) with the back-end replaced by LLVM.

Another way to use LLVM is as a JIT engine. This is done by projects such as Unladen Swallow [3], a JIT-ing Python implementation. The basic idea is to translate the source language code to LLVM IR, and then use LLVM to compile it to native code which is then executed. This is also what the emulator in this thesis does.

Chapter 3

Purpose and Method

3.1 Purpose

When the simple interpreting emulator for the video engine was changed for a JIT-compiling implementation, it provided a significant speed-up. The purpose of this thesis is to investigate what can be done in terms of compiler optimisations to make it run even faster. One reason for implementing the JIT compilation using LLVM was, apart for the convenience of it, that LLVM provides much analysis and many optimisation passes that could be used, and that the infrastructure makes it easy to implement new optimisations. This thesis leverages on that functionality to evaluate, patch, extend and implement new passes to increase performance.

Much JIT research is targeted at deciding which parts of a program to compile and how much effort to use for optimisations in order to provide undisrupted execution, which is necessary for interactive applications. In this case, however, the focus is on net speed-up. It does not matter if the emulator spends a second or two extra on start-up if it means the total execution time is reduced.

3.2 Benchmarks

Optimisations are evaluated by running a number of benchmarks. The benchmarks consist of decoding video files in different formats, as described in Table 3.1.

Benchmark	Format	Resolution	Frames
A	α	720×480	455
B	α	720×526	219
C	β	720×480	455
D	β	1280×720	320

Table 3.1: Benchmarks.

The decoder implementations for the two formats differ quite substantially. Most importantly, the α decoder does most of its work using hardware accel-

erators, while the β decoder is implemented entirely in software. This means that optimisations that speed up firmware execution will have a much greater effect on the β benchmarks. Benchmark C is actually the same video clip as Benchmark A transcoded into the β encoding.

3.3 Measurement Points

As the objective of the optimisations is to make the benchmarks run faster, the time to run benchmarks is measured. However, to determine if an optimisation is actually improving the code, and to evaluate how expensive it is to perform, it is desirable to break down the benchmark time into two parts: compilation time and execution time. This is easily done in a traditional compiler system where code is first compiled and then executed, in separate stages. In this emulator system, the phases are intertwined as LLVM IR cannot be generated for all functions before execution, and machine code is generated lazily as functions are called.

The approach taken is to measure the following:

- Time spent running the emulator
- Time spent building the LLVM IR, excluding optimisations
- Time spent running optimisation passes and generating machine code
- Time spent in hardware emulation.

Those measurements are used to calculate the following values:

- Compilation time, which is the time spent building the LLVM IR plus time spent optimising and generating machine code
- Execution time, which is the time spent running the emulator minus the compilation time.

In some cases, the time spent in hardware emulation is also presented. In tables, such time is presented separately, but in diagrams it is subtracted from the execution time so that the height of the bar represents the total time to run a benchmark. See for example Table 3.2 and Figure 3.1.

3.4 Measurement Implementation

LLVM provides tools for collecting timing information and other statistics. These can be activated for the low-level tools such as `opt` (which is used to run optimisations on LLVM IR) using command-line arguments such as `-stats` and `-time-passes`. The same functionality can be enabled inside the emulator system internally. These switches will make print-outs of how much time is spent in each optimisation and code generation pass, and statistics such as the number of emitted machine instructions.

In addition to this information, additional timers are used for measuring the time building LLVM IR, etc. The timers are implemented using calls to the POSIX function `gettimeofday(2)` and measure wall clock time.

3.5 Equipment

Measurements are run on a computer used solely for this purpose. The computer is a Dell Optiplex GX270 with an Intel Pentium 4 3.00 GHz processor and 1 GB memory. It is running a vanilla install of Debian GNU/Linux 5.0.3, and is in single user mode during tests to make it quiescent. Address space layout randomisation is disabled on the system in an attempt to make it more homogeneous between executions.

The version of LLVM that is used for the benchmarks is revision 89096 (except when the patches discussed in Section 5.1.1 are disabled) of the SVN trunk, compiled with GCC (Debian 4.3.2-1.1) and configured with the options `-enable-jit -enable-targets=host-only -enable-optimized`.

3.6 Presentation of Results

Each measurement is run a number of times (ten, if nothing else is stated) and the mean value, standard deviation and number of measurements are presented. Standard deviation is calculated as:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

where σ is the standard deviation, $x_1, \ldots, x_n$ the samples and $\overline{x}$ the mean of the samples.

Speed-up is defined as

$$S = \frac{T_{old}}{T_{new}}$$

where S is the speed-up, T_{old} time before an optimisation and T_{new} time after optimisation. For example, if a benchmark runs for 15 s before an optimisation and 10 s after, that optimisation is said to give the benchmark a speed-up of 1.5, or 50 %. If the benchmark runs slower with an optimisation, say it runs for 15 s without an optimisation and 20 s with, it is said to give the benchmark a speed-up of 0.75, or a 25 % slowdown. Unless otherwise specified, speed-up refers to benchmark *execution time* throughout the thesis.

3.7 Baseline

The time to execute the benchmarks without optimisations, that is the time it took before this thesis work begun, is used as a baseline against which optimisations are evaluated. Baseline execution times for the benchmarks are presented in Table 3.2 and Figure 3.1, together with times for running the benchmarks in interpreted mode to show what an improvement the move to JIT was for the emulator.

Note how the JIT-mode is about twice as fast for the A and B benchmarks compared to interpretation, and more than ten times faster for the C and D benchmarks. The fact that the β decoder, which is used for the C and D benchmarks, is entirely software based is the reason why interpretation is so slow for those benchmarks, and why they have so much to gain from JIT compilation.

Mode	Benchmark	Execution (s)		Compile (s)		HW (s)	
		mean	σ	mean	σ	mean	σ
Interpreted	A	492.79	0.41	n.a		169.17	0.17
	B	243.91	0.35	n.a		87.28	0.17
	C	2,588.58	2.18	n.a		0.00	0.00
	D	4.590.85	4.03	n.a		0.00	0.00
JIT	A	246.13	0.41	12.25	0.06	165.36	0.05
	B	123.66	0.17	12.28	0.08	85.46	0.09
	C	188.06	0.11	11.07	0.08	0.00	0.00
	D	335.11	0.13	9.99	0.06	0.00	0.00

Table 3.2: Baseline and interpreted benchmark times.

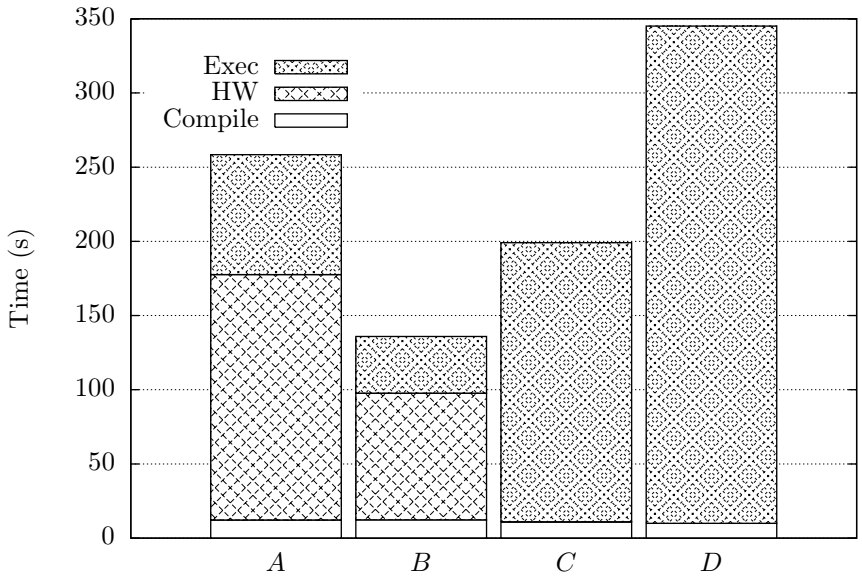

Figure 3.1: Baseline compilation, hardware and execution times.

Chapter 4

The Emulator System

4.1 System Overview

The RASC processor is a processor developed specifically for use in the video engine. The emulation of the processor, which is what this thesis is concerned with, is part of a larger emulation system; see Figure 4.1.

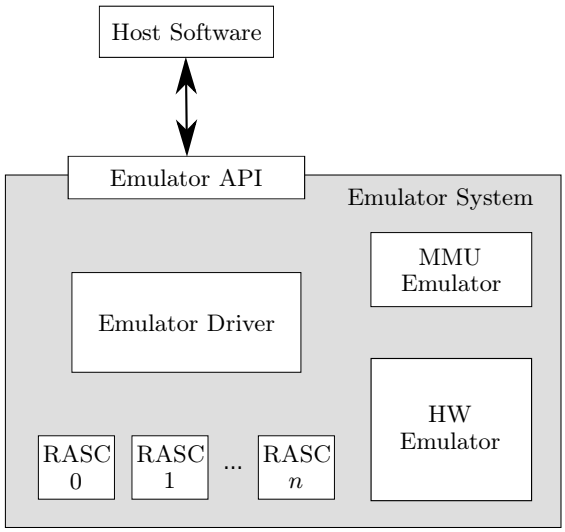

Figure 4.1: The emulator system.

In a deployed system, a program running on a host CPU uses the system to encode or decode video. Special video coding software runs on one or more RASC CPUs, and hardware accelerators are used for part of their work. These components, the host CPU, RASC CPUs and hardware accelerators, communicate using shared memory and interrupts.

When run against the emulator, the host program utilises an API which is equivalent to the one it would use if it were talking to real hardware. The emulator should, as closely as possible, mimic the hardware system.

The RASC emulators read a stream of instructions and executes them, updating their registers and the shared memory.

The MMU (memory management unit) Emulator is important for making the shared-memory communication work. It determines if an address in a load or store operation belongs to RASC memory (pointing to the address of an instruction or something on the execution stack), to memory shared with the host program (the host program has to register such memory areas with the MMU), or if the address is a special hardware register and the access should trigger some action in the HW Emulator.

The HW Emulator contains software implementations of all the hardware accelerator blocks.

Orchestrating the process of emulation, the Emulator Driver steps the RASC emulators in a round-robin fashion and handles the communication between the different components of the emulator and the host software.

4.2 RASC Emulation Overview

The flow from firmware source to execution is illustrated in Figure 4.2.

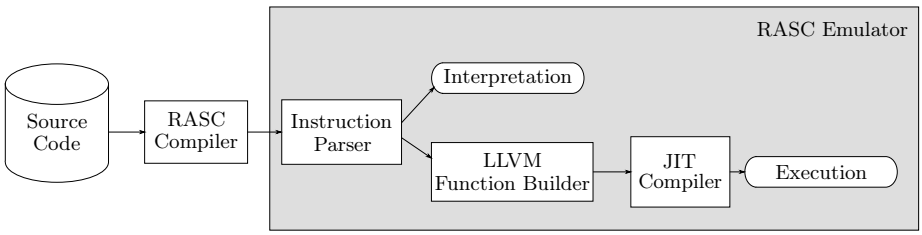

Figure 4.2: Flow from RASC source code to execution.

RASC source is compiled using a custom compiler. The resulting file is referred to as the RASC firmware. This firmware is optimised for fast RASC execution and small code size. The emulator begins execution from the first byte of the file.

An instruction parser in the emulator parses the instructions in the firmware file. Depending on emulation mode, the instructions can then be run in an interpreter straight away: the interpreter simply looks at the current instruction, and performs some action accordingly.

For JIT compilation, the RASC instruction stream is translated to LLVM instructions. This is not a simple one-to-one translation: the details are explained in Section 4.3.2 below. The LLVM instructions are then JIT-compiled to native machine code and executed.

4.3 LLVM Representation

4.3.1 Building Blocks

LLVM represents programs as modules, functions, basic blocks, instructions and values. These are described in detail in [2].

The concept of modules is similar to translation units in C. A module contains functions and global variables. The RASC emulator only produces one module, representing the entire firmware.

Functions are similar to function concepts in most programming languages: they take arguments, return values and contain instructions to be executed when the function is called.

In functions, the instructions are stored in basic blocks. A basic block contains an ordered list of non-branching instructions and a terminating branch or return instruction. This terminating instruction determines which basic blocks can be successors in the execution. The basic blocks of a function form a graph which is known as the control-flow graph (CFG). One basic block is the function entry block, and does not have preceding blocks. Execution of a function flows from the entry block to a block with a return as terminating instruction.

An LLVM value can be a literal, such as 42, or an instruction such as an add instruction. LLVM programs are on static single assignment (SSA) form, meaning a value is defined exactly once. This makes it easy to find definitions and uses of values and work with them.

4.3.2 RASC Emulation Functions

The RASC emulator builds RASC emulation functions with names such as `Func`*addr*, which take as argument a pointer to a variable representing the RASC CPU state. Running such a function has the effect of emulating firmware execution starting at address *addr*.

Instructions in the function object operate on the state variable which is passed when calling the function. For example, a RASC instruction which adds the values of two registers and stores it in a third would result in LLVM instructions to load the register values from the state variable, add the values and write the result back to the state variable.

Memory accessing instructions in firmware cannot be easily translated to memory accessing LLVM instructions. An address in the RASC address space is generally not the same as an address in host space. The address could represent a hardware register, in which case a write should trigger some action by the HW emulator. Otherwise, the address should be translated to an address in the host memory. Both these tasks are handled by the MMU emulator, which is reached via calls to functions called `emul_read32`, `emul_write32` and `emul_translate_addr`.

Function objects are generally built to match functions in the firmware. Calling such a function object and passing a pointer to a state variable has the same effect on the state as running the firmware function on a RASC with the same state. Such a function always ends with a return of zero.

There are however situations when an LLVM function does not match a firmware function. For example, the destination of an indirect jump instruction (where the target address is read from a register) is normally not known when the function object is built, as it is determined at run-time. In such situations, the function builder does not know how to continue, and thus terminates the function with a return instruction which returns the special value one.

The same thing happens when the emulation should stop for some reason. This is the case with the `wait` instruction, which means the emulator should

temporarily suspend emulation, check for an interrupt, and then resume emulation. The situation is handled in the same way as above: by a return instruction which returns one.

This means that the generated functions can return for two reasons: either because there was a return instruction in the RASC program, or because the emulator cannot or will not continue in that function. The function's return value is used to separate these cases. A return value of zero means it is a normal return and a return value of one means it is a get-out-of-emulation return. This affects how function calls are emulated. The calling function must look at the return value, and if the return value is one, the calling function must propagate that value by returning one itself.

4.4 Stepping the Emulator

When it comes to running a RASC emulator, the emulator driver calls a function, `step_jit()`, passing the state of the CPU as an argument.

The emulator looks at the program counter (PC) in the state variable to determine where to start execution. This address is zero the first time the CPU is stepped. Using the address, the emulator looks up the function object that has been built to emulate execution starting at this address. If there is no such function object, one is built as described above.

To execute the function, the emulator calls on LLVM's `ExecutionEngine` to get a function pointer to the function. If the function object has not been compiled before, it is now compiled, and native code is emitted into the host computer's memory. A function pointer into this memory is returned, and the emulator makes a call through the pointer.

LLVM call instructions are compiled into native call instructions which point either to the natively compiled functions in memory or, if the called function is not compiled yet, to special functions which compile the called functions and back-patch the address into the call sites. This way, if a function object is never called, native code is never generated for it. That is why it is called JIT compilation: a part of a program is compiled just in time for its execution.

4.5 An Example

The flow from RASC source code to native machine instructions is illustrated in this example.

A simple function is defined in the firmware, and a call to it is inserted at the very beginning of the program, see Listing 4.1. After compilation into RASC machine code, the function looks as in Listing 4.2 The LLVM Function object is shown in Listing 4.3, and the machine code emitted after JIT compilation of the function object is shown in Listing 4.4.

(After studying the generated code, the careful reader may now start to realise the potential for optimisations of the code.)

```
#define NFIBS   100

unsigned fibs[NFIBS];

void fib(void)
{
    int         i;

    fibs[0] = 0;
    fibs[1] = 1;

    for (i = 2; i < NFIBS; i++) {
        fibs[i] = fibs[i-1] + fibs[i-2];
    }
}

void rasc0_start(void)
{
    fib();
    ...
```

Listing 4.1: `fib` implemented in the firmware source code.

```
<fib>:
lpc     r0,2360c <fibs>
mov     r1,0
st      r1,[r0,0]
mov     r2,r0
mov     r1,1
st      r1,[r0,4]
add     r2,4
mov     r3,0x62

<.L4>:
ld      r0,[r2,0xfffffffc]
ld      r1,[r2,0]
add     r0,r1
st      r0,[r2,4]
add     r2,4
dbne    r3,5681 <.L4>
jmp     lr
```

Listing 4.2: `fib` function in RASC machine code.

```
define internal i32 @Func0000566d(%0* noalias nocapture) {
  %2 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 0
  %3 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 1
  %4 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 2
  %5 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 3
  %6 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 14
  %7 = getelementptr inbounds %0* %0, i32 0, i32 0
  store i32 144908, i32* %2
  call void @emul_write32(i32 144908, i32 0, i8* null)
  %8 = load i32* %2
  store i32 %8, i32* %4
  store i32 1, i32* %3
  %9 = load i32* %2
  %10 = add i32 %9, 4
  call void @emul_write32(i32 %10, i32 1, i8* null)
  %11 = load i32* %4
  %12 = add i32 %11, 4
  store i32 %12, i32* %4
  store i32 98, i32* %5
  br label %15

; <label>:13
  %14 = load i32* %6
  store i32 %14, i32* %7
  ret i32 0

; <label>:15
  %16 = load i32* %4
  %17 = add i32 %16, -4
  %18 = call i32 @emul_read32(i32 %17, i8* null)
  store i32 %18, i32* %2
  %19 = load i32* %4
  %20 = call i32 @emul_read32(i32 %19, i8* null)
  store i32 %20, i32* %3
  %21 = load i32* %2
  %22 = add i32 %21, %20
  store i32 %22, i32* %2
  %23 = load i32* %4
  %24 = add i32 %23, 4
  call void @emul_write32(i32 %24, i32 %22, i8* null)
  %25 = load i32* %4
  %26 = add i32 %25, 4
  store i32 %26, i32* %4
  %27 = load i32* %5
  %28 = add i32 %27, -1
  store i32 %28, i32* %5
  %29 = icmp eq i32 %28, 0
  br i1 %29, label %13, label %15
}
```

Listing 4.3: LLVM function object to emulate execution of the `fib` function.

```
0xb6976ee0:     push   esi
0xb6976ee1:     sub    esp,0x10
0xb6976ee4:     mov    esi,DWORD PTR [esp+0x18]
0xb6976ee8:     mov    DWORD PTR [esp+0x8],0x0
0xb6976ef0:     mov    DWORD PTR [esp+0x4],0x0
0xb6976ef8:     mov    DWORD PTR [esp],0x2360c
0xb6976eff:     mov    DWORD PTR [esi+0x4],0x2360c
0xb6976f06:     call   0xb73e3520 <emul_write32>
0xb6976f0b:     mov    eax,DWORD PTR [esi+0x4]
0xb6976f0e:     mov    DWORD PTR [esi+0xc],eax
0xb6976f11:     mov    DWORD PTR [esi+0x8],0x1
0xb6976f18:     mov    eax,DWORD PTR [esi+0x4]
0xb6976f1b:     add    eax,0x4
0xb6976f1e:     mov    DWORD PTR [esp],eax
0xb6976f21:     mov    DWORD PTR [esp+0x8],0x0
0xb6976f29:     mov    DWORD PTR [esp+0x4],0x1
0xb6976f31:     call   0xb73e3520 <emul_write32>
0xb6976f36:     add    DWORD PTR [esi+0xc],0x4
0xb6976f3a:     mov    DWORD PTR [esi+0x10],0x62
0xb6976f41:     mov    eax,DWORD PTR [esi+0xc]
0xb6976f44:     add    eax,0xfffffffc
0xb6976f47:     mov    DWORD PTR [esp],eax
0xb6976f4a:     mov    DWORD PTR [esp+0x4],0x0
0xb6976f52:     call   0xb73e34a0 <emul_read32>
0xb6976f57:     mov    DWORD PTR [esi+0x4],eax
0xb6976f5a:     mov    eax,DWORD PTR [esi+0xc]
0xb6976f5d:     mov    DWORD PTR [esp],eax
0xb6976f60:     mov    DWORD PTR [esp+0x4],0x0
0xb6976f68:     call   0xb73e34a0 <emul_read32>
0xb6976f6d:     mov    DWORD PTR [esi+0x8],eax
0xb6976f70:     add    eax,DWORD PTR [esi+0x4]
0xb6976f73:     mov    DWORD PTR [esi+0x4],eax
0xb6976f76:     mov    DWORD PTR [esp+0x4],eax
0xb6976f7a:     mov    ecx,DWORD PTR [esi+0xc]
0xb6976f7d:     add    ecx,0x4
0xb6976f80:     mov    DWORD PTR [esp],ecx
0xb6976f83:     mov    DWORD PTR [esp+0x8],0x0
0xb6976f8b:     call   0xb73e3520 <emul_write32>
0xb6976f90:     add    DWORD PTR [esi+0xc],0x4
0xb6976f94:     mov    eax,DWORD PTR [esi+0x10]
0xb6976f97:     dec    eax
0xb6976f98:     mov    DWORD PTR [esi+0x10],eax
0xb6976f9b:     test   eax,eax
0xb6976f9d:     jne    0xb6976f41
0xb6976fa3:     mov    eax,DWORD PTR [esi+0x3c]
0xb6976fa6:     mov    DWORD PTR [esi],eax
0xb6976fa8:     xor    eax,eax
0xb6976faa:     add    esp,0x10
0xb6976fad:     pop    esi
0xb6976fae:     ret
```

Listing 4.4: Machine code after JIT compilation of the `fib` function.

Chapter 5

Optimisations

5.1 Removing Redundant Memory Accesses

5.1.1 Improving Alias Analysis

Alias analysis is the process of figuring out whether two pointers may point to the same object in memory or not. Such information is vital for compiler optimisations such as dead load elimination and dead store elimination.

The basic alias analysis implementation (there are others, but this one is good enough for the emulator) in LLVM was improved to handle two cases commonly occurring in the IR generated by the emulator.

Null-pointers

The MMU emulator functions each take a call-back function as parameter. In the LLVM IR, this will always be null, and it thus seems obvious that it cannot alias with anything. However, this heuristic was not implemented in LLVM, and in code such as this:

```
%t = type { i32 }
declare void @test1f(i8*)

define void @test1(%t* noalias %stuff ) {
    %p = getelementptr inbounds %t* %stuff, i32 0, i32 0
    %before = load i32* %p
    call void @test1f(i8* null)
    %after = load i32* %p ; <--- This should be a dead load
    %sum = add i32 %before, %after;

    store i32 %sum, i32* %p
    ret void
}
```

LLVM would be unsure if the null pointer and `%p` could alias, so that the call to `@test1f` might modify the variable that `%p` pointed to. As it was unsure about this, it did not consider the second load redundant, and failed to remove it as a dead load.

A patch which adds the heuristic that null pointers do not alias with anything (not even themselves, as one cannot read or write through them) was added to the LLVM trunk in revision 86582, see B.1.

Constant Pointers

Another thing which occurs frequently in the generated LLVM IR is emulated accesses to fixed addresses, for example hardware registers. This code provides an example:

```
store i32 -268300288, i32* %2
%79 = call i32 @emul_read32(i32 -268300276, i8* null)
store i32 %79, i32* %6
%80 = load i32* %2; <-- This should be a dead load
%81 = add i32 %80, 8
```

The variable `%2` is derived from the pointer to the state variable, which is marked `noalias`. The problem here is that LLVM is unsure whether or not the constant in the `@emul_read` call might alias with `%2`, and cannot deduce that the load from `%2` is dead because it has not been written to since the store at the first line.

As the `noalias` attribute means that the parameter is *the only way* to access the underlying variable from the function, it can be safely assumed that a constant address in the function cannot point to the same variable. A patch that covers this case was added to the LLVM trunk in revision 88760, see B.2.

Results

Alias analysis is vital to dead load and store elimination. Table 5.1 shows the number of dead loads and stores (using the default global value numbering and dead store elimination passes) before and after the improvements to alias analysis.

Benchmark	Un-improved AA		Improved AA	
	Dead loads	Dead Stores	Dead Loads	Dead Stores
A	5,148	1,147	14,380	3,658
B	5,148	1,147	14,380	3,658
C	4,530	1,092	10,558	2,512
D	4,200	1,005	9,559	2,343

Table 5.1: Dead loads and stores with and without improved alias analysis.

This improvement has no effect on the execution of the baseline, as dead load elimination is not enabled there, but it is vital for making the other optimisations effective. In the rest of this thesis, measurements are conducted with these patches applied.

```
1  define internal i32 @Func0000566d(%0* noalias nocapture) {
2    %2 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 0
3    %3 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 1
4    %4 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 2
5    %5 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 3
6    %6 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 14
7    %7 = getelementptr inbounds %0* %0, i32 0, i32 0
8    store i32 144908, i32* %2
9    call void @emul_write32(i32 144908, i32 0, i8* null)
10   %8 = load i32* %2
11   store i32 %8, i32* %4
12   store i32 1, i32* %3
13   %9 = load i32* %2
14   %10 = add i32 %9, 4
15   call void @emul_write32(i32 %10, i32 1, i8* null)
16 [snip]
```

Listing 5.1: Part of the `fib` function with several dead loads.

```
1  define internal i32 @Func0000566d(%0* noalias nocapture) {
2  ; <label>:1
3    %2 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 0
4    %3 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 1
5    %4 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 2
6    %5 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 3
7    %6 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 14
8    %7 = getelementptr inbounds %0* %0, i32 0, i32 0
9    store i32 144908, i32* %2
10   call void @emul_write32(i32 144908, i32 0, i8* null)
11   store i32 1, i32* %3
12   call void @emul_write32(i32 144912, i32 1, i8* null)
```

Listing 5.2: Part of the `fib` function with dead loads eliminated.

5.1.2 Dead Load Elimination

Dead load elimination, sometimes called redundant load elimination, is the elimination of loads which are unnecessary because the result of the load is already known. The result of a load can be known beforehand either because a load from the same address has been made before, or a store to the address has been made before.

Dead loads are very frequent in the code generated by the emulator. RASC instructions operate on registers, which means that when LLVM IR is generated to emulate an instruction, one or more loads from the state variable are produced, many of which are redundant. This is illustrated in Listing 5.1 where, for example, there are two loads from `%2` even though none is necessary as the variable is not touched after the store on line 8 and that value could simply be propagated to where it is needed.

LLVM performs dead load elimination as part of its global value numbering (GVN) pass. The effect on the example code is shown in Listing 5.2, and the effect on the benchmarks is shown in Table 5.2. There is not much speed-up on the α benchmarks, but for the β benchmarks, this optimisation provides a 4 % speed-up.

For the α benchmarks, the low execution time reduction weighed against the increased compilation time makes the optimisation seem counter-productive. However, this optimisation is crucial for getting more fruitful optimisations to work, and therefore worth the extra compilation time.

Benchmark	Execution (s)		Compile (s)		Dead Loads	Speed-up
	mean	σ	mean	σ		
A	244.88	0.46	13.72	0.08	14,380	1.01
B	123.43	0.10	13.75	0.07	14,380	1.00
C	181.33	0.08	12.45	0.04	10,558	1.04
D	321.24	0.09	11.17	0.05	9,559	1.04

Table 5.2: Benchmarks run with global value numbering.

5.1.3 Dead Store Elimination

A store is dead if the value it writes is never read before a subsequent store instruction writes to the same memory location. It is also dead if it writes the same value that was loaded from the memory location earlier, and there has been no stores in between.

Dead stores are common in the generated IR for the same reasons as dead loads. The only stores that are really necessary are the ones which affect the state variable which is seen after return from the function, or by called functions which are passed a pointer to the state variable.

LLVM provides a pass for dead store elimination (DSE), which was enabled in the emulator. However, several dead stores were still present because the pass only performs DSE within basic blocks.

It is easy to see if a store is followed by another store within a basic block: they just have to come in successive order in the instruction list. And if there are no loads or other uses of the location they store to in between, then the first store is dead.

The situation is more complex between basic blocks, as execution may take different paths between the instructions. To remove the first store, one must be positive that the second store will always be executed afterwards. This happens if the second store *post-dominates* the first. A basic block B post-dominates A if every path from A to the exit of the function passes through B [4, 10.4.1].

LLVM provides an analysis called MemoryDependenceAnalysis. This pass answers the question of what instruction a memory-accessing instruction depends on. For example, a load from location p may be dependent on a preceding store to that location which would define the value at p, an allocation instruction which would make the value undefined or a call instruction which could have effect on the value, making it unknown.

For dead store elimination however, the opposite question is more relevant: what instructions are depending on a given store instruction. LLVM does not provide such analysis (it could have been called ReverseMemoryDependenceAnalysis, and it might happen one day), but the necessary information can still be obtained by simply walking through all instructions, see if they depend on a store and count how many dependencies there are on each store. This is shown in Algorithm 5.1 and the code in B.3.

This backwards approach to finding the number of uses of a store is not efficient, and was therefore not submitted to LLVM. However, the emulator's benefit from global DSE is large enough that this implementation is better than

```
uses[I] = 0 ∀ I ∈ F
for each instruction I ∈ F do
  if I is a call or load instruction then
    for each instruction I_D that I depends on do
      if I_D is a store instruction then
        uses[I_D] ← uses[I_D] + 1
      end if
    end for
  end if
end for
```

Algorithm 5.1: Number of uses of each store instruction in function F.

none at all.

Post-dominance analysis and information about number of uses of stored values is sufficient for deciding if a store is dead because of a subsequent store or not. Algorithm 5.2 and the code in B.4 show global dead store elimination added to the DSE pass used by the emulator.

```
D ← instructions that a store instruction I_S depends on
if |D| = 1 then
  I_D ← the instruction in D
  if I_D is a load from same location as I_S and the value stored by I_S is I_D
  then
    Remove I_S as a dead store
  end if
end if
for each I_D ∈ D do
  if I_D is a store instruction then
    if uses[I_D] = 0 then
      if block(I_S) post dominates block(I_D) then
        if storage type of I_D ≤ storage type of I_S then
          Remove I_D as a dead store
        end if
      end if
    end if
  end if
end for
```

Algorithm 5.2: Dead store removal with non-local dependencies.

Benchmark performance when running with this improved DSE together with GVN is shown in Table 5.3. The number of dead stores is up by between 50 and 70 % (compare Table 5.1), and the β benchmarks get a nice speed-up, while the α benchmarks do not benefit much, but rather suffer from the increased compilation time. Because of the benefit on the β benchmarks, this optimisation will continue to be used.

Benchmark	Execution (s)		Compile (s)		Dead Stores	Speed-up
	mean	σ	mean	σ		
A	244.14	0.22	14.60	0.09	5,763	1.01
B	123.37	0.27	14.59	0.11	5,763	1.00
C	171.88	0.08	13.17	0.07	4,470	1.09
D	302.47	0.11	11.84	0.06	4,097	1.11

Table 5.3: Benchmarks run with GVN and improved DSE.

5.2 Reducing MMU Emulation

Each load and store in the RASC firmware generates calls to MMU emulator functions in the LLVM IR. The generated calls are either for `emul_read32`, `emul_write32` or `emul_translate_addr`. These functions look at the address and determines if it represents a hardware register, in which case the HW emulator takes action, or else it looks in a structure of memory mappings and translates the address to a host address which it reads or writes to.

Loads and stores are frequent in RASC programs, as in most programs, and as each call to the MMU emulator is much more costly than a native load or store, this is considerable overhead. The numbers of MMU calls are shown in Table 5.4. (Total instruction counts are available in Appendix A.) It is desirable to replace as many of these instructions as possible with native loads and stores.

Benchmark	`emul_read32`	`emul_write32`	`emul_translate_addr`
A	2,279	2,134	1,210
B	2,279	2,134	1,210
C	589	898	2,060
D	536	801	1,771

Table 5.4: Number of MMU calls in benchmarks.

5.2.1 Compile-time Constant Address Translation

If the address in a call to the MMU Emulator is constant, the address translation can be done compile-time. Such cases are not uncommon. For example, statically allocated data areas have static addresses pointing into the firmware, as in the case of the `fibs` array from Listing 4.1.

An optimisation pass for doing translation of constant addresses compile-time was already implemented along the lines of Algorithm 5.3. Addresses referring to hardware registers cannot be translated, but the rest can.

This seems straight-forward enough, but as it was it did not have any impact on the emulator's performance. The reason is that accesses to constant addresses look something like this (from the `fib` example in Listing 4.2):

```
<fib>:
lpc    r0,2360c <fibs>
mov    r1,0
st     r1,[r0,0]
mov    r2,r0
mov    r1,1
st     r1,[r0,4]
```

It turns out that the address operands to the store instructions are not constant, but rather the result of retrieving a value from `r0` and adding with constant values (0 and 4 in this case). Dead load elimination is needed to propagate the constants into the arguments of the calls to `emul_write32`. With the GVN pass enabled, this optimisation removes a significant number of MMU calls and gives the benchmarks a speed-up, as seen in Table 5.5.

for each call instruction $I_C \in F$ **do**
  $A \leftarrow$ first function argument (the address)
  if $callee(I_C) =$ `emul_read32` and A is constant **then**
    if A can be translated to a host address **then**
      Replace I_C with a read from $A_{translated}$
    end if
  else if $callee(I_C) =$ `emul_write32` and A is constant **then**
    if A can be translated to a host address **then**
      Replace I_C with a write to $A_{translated}$
    end if
  else if $callee(I_C) =$ `emul_translate_addr` and A is constant **then**
    if A can be translated to a host address **then**
      Replace I_C with $A_{translated}$
    end if
  end if
end for

Algorithm 5.3: Compile-time constant address translation for function F.

Benchmark	Execution (s)		Compile (s)		Removed MMU Calls	Speed-up
	mean	σ	mean	σ		
A	240.02	0.21	14.34	0.07	1,001	1.03
B	121.17	0.18	14.37	0.11	1,001	1.02
C	133.17	0.26	12.62	0.07	1,232	1.41
D	233.70	0.39	11.38	0.09	1,056	1.43

Table 5.5: Performance with compile-time translation of constant addresses.

5.2.2 Moving Address Translation Out of Loops

Listing 5.3 shows an example piece of RASC code that accesses memory in loops, and Listing 5.4 shows the resulting LLVM IR. Such loops are heavily dominated by the calls to MMU emulation. Since the accesses in a loop are all to the same array, it would be desirable to just translate the base address of the array, and then use that and the array offsets to compute host addresses at each iteration.

To detect loop accesses to arrays, an optimisation pass has to look at the address argument of MMU calls in loops, and analyse how they vary between loop iterations. LLVM provides an analysis that does this: ScalarEvolutionAnalysis. The pass looks for addresses that can be represented as *add recurrences*. This is a concept in scalar evolution which is based on [6]. Scalar evolution provides utilities for building expressions for the address at the first loop iteration, A_{base}, which is loop invariant.

Those expressions can be used to get the translated address as

$$T(A) = T(A_{base}) + A - A_{base}$$

where $T(A_{base})$ is the address in the first iteration, translated to a host address.

```
static void block_zero(void)
{
    int i,j;
    for (j=0;j<6;j++)
        for (i=0;i<64;i++) g_mb->QF[j][i] = 0;
}

<block_zero>:
push   lr-r14

<.LCFI14>:
lpc    r3,87e0 <mb>
mov    r4,6

<.L160>:
mov    r0,0
mov    r2,2
mov    r1,0x3f
sth    r0,[r3,0]

<.L161>:
mov    r0,r2
add    r0,r3
mov    lr,0
sth    lr,[r0,0]
add    r2,2
dbne   r1,26db <.L161>
add    r3,0x80
dbne   r4,26d2 <.L160>
popr   lr-r14
```

Listing 5.3: `block_zero` as implemented in RASC firmware.

Since A_{base} is loop invariant, the expression for $T(A_{base})$ can be moved outside the loop. The translated address, $T(A)$, can then be used to replace the MMU emulation call inside the loop. This is outlined in Algorithm 5.4, which is an optimisation pass that is applied to loops. The code is shown in B.5.

The optimisation has to be careful when moving translations out of a loop. The new translation is only allowed to execute in cases where an MMU call would have been performed inside the loop, otherwise it might alter the program's behaviour. Therefore, translations moved out of loops are conditional on that the loop's trip count is known to be greater than zero. Also, there may be different execution paths through a loop body, so knowing that a loop is executed is not enough to know an MMU call inside it will be made. A condition that the MMU call post-dominates the loop header is required for this optimisation.

Listing 5.5 shows LLVM IR for the `block_zero` function after this optimisation. Note that there are no calls for MMU emulation inside the loops. Also note that emulation calls have been removed both from the inner and outer loop, because the optimisation is applied to inner loops first. Since the base address of the array is a constant, it has been translated to a host address (134769016) at compile time.

Table 5.6 shows benchmark performance together with number of translations moved out of loops. GVN, improved DSE and compile-time translation of constant addresses are also enabled. This optimisation provides a nice speed-up for the β benchmarks, and also for the α benchmarks, although not as significant as it runs many loops in hardware emulation instead of software. Note that the optimisation is cheap in terms of additional compilation time; it is definitely worth the extra effort.

```
define internal i32 @Func000026c9(%0* noalias nocapture) {
; <label>:1
  %2 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 0
  %3 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 1
  %4 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 2
  %5 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 3
  %6 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 4
  %7 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 14
  %8 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 15
  %9 = getelementptr inbounds %0* %0, i32 0, i32 0
  %10 = load i32* %8
  %11 = add i32 %10, -4
  %12 = lshr i32 %11, 2
  %13 = getelementptr i32* inttoptr (i32 134734176 to i32*), i32 %12
  %14 = load i32* %7
  store i32 %14, i32* %13
  store i32 %11, i32* %8
  store i32 34784, i32* %5
  store i32 6, i32* %6
  br label %30

; <label>:15
  %16 = add i32 %32, 128
  store i32 %16, i32* %5
  %17 = add i32 %31, -1
  store i32 %17, i32* %6
  %18 = icmp eq i32 %17, 0
  br i1 %18, label %28, label %30

; <label>:19
  %20 = phi i32 [ %26, %19 ], [ 63, %30 ]
  %21 = phi i32 [ %25, %19 ], [ 2, %30 ]
  %22 = add i32 %21, %32
  store i32 %22, i32* %2
  store i32 0, i32* %7
  %23 = call i8* @emul_translate_addr(i32 %22, i32 1, i8* null)
  %24 = bitcast i8* %23 to i16*
  store i16 0, i16* %24
  %25 = add i32 %21, 2
  store i32 %25, i32* %4
  %26 = add i32 %20, -1
  store i32 %26, i32* %3
  %27 = icmp eq i32 %26, 0
  br i1 %27, label %15, label %19

; <label>:28
  %29 = load i32* %13
  store i32 %29, i32* %7
  store i32 %10, i32* %8
  store i32 %29, i32* %9
  ret i32 0

; <label>:30
  %31 = phi i32 [ %17, %15 ], [ 6, %1 ]
  %32 = phi i32 [ %16, %15 ], [ 34784, %1 ]
  store i32 0, i32* %2
  store i32 2, i32* %4
  store i32 63, i32* %3
  %33 = call i8* @emul_translate_addr(i32 %32, i32 1, i8* null)
  %34 = bitcast i8* %33 to i16*
  store i16 0, i16* %34
  br label %19
}
```

Listing 5.4: LLVM IR generated for the `block_zero` function.

```
define internal i32 @Func000026c9(%0* noalias nocapture) {
; <label>:1
  %2 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 0
  %3 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 1
  %4 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 2
  %5 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 3
  %6 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 4
  %7 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 14
  %8 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 15
  %9 = load i32* %8
  %10 = add i32 %9, -4
  %11 = lshr i32 %10, 2
  %12 = getelementptr i32* inttoptr (i32 134734232 to i32*), i32 %11
  %13 = load i32* %7
  store i32 %13, i32* %12
  store i32 %10, i32* %8
  store i32 34784, i32* %5
  store i32 6, i32* %6
  br label %22

; <label>:14
  store i32 %tmp13, i32* %5
  store i32 %tmp15, i32* %6
  %indvar.next4 = add i32 %indvar3, 1
  %exitcond9 = icmp eq i32 %indvar.next4, 6
  br i1 %exitcond9, label %19, label %22

; <label>:15
  %indvar = phi i32 [ %indvar.next, %15 ], [ 0, %22 ]
  %tmp1 = sub i32 62, %indvar
  %tmp2 = shl i32 %indvar, 1
  %tmp7 = add i32 %tmp2, %tmp6
  %tmp8 = add i32 %tmp2, 4
  store i32 %tmp7, i32* %2
  store i32 0, i32* %7
  %16 = sub i32 %tmp7, %tmp17
  %17 = add i32 %16, %23
  %18 = inttoptr i32 %17 to i16*
  store i16 0, i16* %18
  store i32 %tmp8, i32* %4
  store i32 %tmp1, i32* %3
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp eq i32 %indvar.next, 63
  br i1 %exitcond, label %14, label %15

; <label>:19
  %20 = getelementptr inbounds %0* %0, i32 0, i32 0
  %21 = load i32* %12
  store i32 %21, i32* %7
  store i32 %9, i32* %8
  store i32 %21, i32* %20
  ret i32 0

; <label>:22
  %indvar3 = phi i32 [ %indvar.next4, %14 ], [ 0, %1 ]
  %tmp16 = shl i32 %indvar3, 7
  %tmp17 = add i32 %tmp16, 34786
  %23 = add i32 %tmp16, 134769018
  %tmp10 = shl i32 %indvar3, 7
  %tmp6 = add i32 %tmp10, 34786
  %tmp13 = add i32 %tmp10, 34912
  %tmp15 = sub i32 5, %indvar3
  store i32 0, i32* %2
  store i32 2, i32* %4
  store i32 63, i32* %3
  %24 = add i32 %tmp10, 134769016
  %25 = inttoptr i32 %24 to i16*
  store i16 0, i16* %25, align 8
  br label %15
}
```

Listing 5.5: `block_zero` with address translation moved out of loops.

for each MMU emulation call I_C in loop L **do**
  $A \leftarrow$ the address operand of I_C
  if $TC(L)$ (trip count) is not available **then**
    break
  end if
  if I_C does not post-dominate loop header **then**
    continue
  end if
  if A can be represented as an add recurrence by scalar evolution analysis **then**
    $A_{base} \leftarrow$ expression for A at first iteration, inserted in loop header
    $T(A_{base}) \leftarrow$ insert translation of A_{base} to host address in loop header, conditional on $TC(L) > 0$
    $T(A) \leftarrow T(A) + (A - A_{base})$
    Replace I_C with native read, write, etc. using $T(A)$
  end if
end for

Algorithm 5.4: Moving address translations out of loops.

Benchmark	Execution (s)		Compile (s)		Moved Translations	Speed-up
	mean	σ	mean	σ		
A	232.98	0.12	14.67	0.11	135	1.06
B	117.79	0.20	14.67	0.09	135	1.05
C	116.32	0.09	12.94	0.06	457	1.62
D	207.57	0.09	11.70	0.05	383	1.61

Table 5.6: Benchmark performance with address translation moved out of loops.

5.2.3 Combining Translation of Close Addresses

Listing 5.6 shows an example of a function that accesses two elements of a struct. The LLVM IR for the function, shown in Listing 5.7, shows two calls for MMU emulation: one for each write instruction. As with the array accesses in Section 5.2.2, it would be desirable to only perform one address translation when the addresses refer to the same structure.

The approach taken by this optimisation is to look for a set of memory accesses with addresses $\mathcal{A} = \{a_i | a_i = base + x_i\}$ where $base$ is a common base, and x_i are constant offsets. The RASC ABI specifies that there must be at least one memory page of unused address space between each memory mapping, so if the maximum distance between any two addresses in $\mathcal{A}$ is less than this gap, then they must all refer to the same memory mapping.

It is important that the optimisation stays clear of addresses to hardware registers. Hardware register addresses are always constant values, and they are not allowed to be passed as arguments to RASC functions. Base addresses can be restricted to only be addresses that come from loading the value of a RASC

```
struct foo_t {
    int          a;
    int          b;
} foos[100];

void foo(struct foo_t *f)
{
    int          i;

    f->a = 0;
    f->b = 1;
}

<foo>:
mov     r1,1
st      r1,[r0,4]
mov     r1,0
st      r1,[r0,0]
jmp     lr
```

Listing 5.6: The `foo` function as implemented in the RASC firmware.

```
define internal i32 @Func00005663(%0* noalias nocapture) {
  %2 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 0
  %3 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 1
  %4 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 14
  %5 = getelementptr inbounds %0* %0, i32 0, i32 0
  %6 = load i32* %2
  %7 = add i32 %6, 4
  call void @emul_write32(i32 %7, i32 1, i8* null)
  store i32 0, i32* %3
  call void @emul_write32(i32 %6, i32 0, i8* null)
  %8 = load i32* %4
  store i32 %8, i32* %5
  ret i32 0
}
```

Listing 5.7: LLVM IR generated for the `foo` function.

```
define internal i32 @Func00005663(%0* noalias nocapture) {
  %2 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 0
  %3 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 1
  %4 = getelementptr inbounds %0* %0, i32 0, i32 1, i32 14
  %5 = getelementptr inbounds %0* %0, i32 0, i32 0
  %6 = load i32* %2
  %7 = call i8* @emul_translate_addr(i32 %6, i32 4, i8* null)
  %8 = ptrtoint i8* %7 to i32
  %9 = add i32 %8, 4
  %10 = inttoptr i32 %9 to i32*
  store i32 1, i32* %10
  store i32 0, i32* %3
  %11 = bitcast i8* %7 to i32*
  store i32 0, i32* %11
  %12 = load i32* %4
  store i32 %12, i32* %5
  ret i32 0
}
```

Listing 5.8: LLVM IR generated for the `foo` function after optimisation.

register. But what if a constant address is stored in a register and then used as base address of a memory access? Then the constant address will have been constant-propagated by the GVN pass so this does not happen.

A dangerous situation occurs when a RASC function is represented by more than one LLVM function object. This could happen because the firmware contains a `wait` instruction, in which case there will be LLVM functions to emulate the firmware before and after the `wait`. A constant may be stored in a register in the first function, and loaded in the second. Global value numbering only works within functions, so the value is not constant-propagated. Because of this, combining translation of close addresses is only done on functions that are considered "proper", meaning that they correspond to whole RASC functions, or at least the first part.

When inserting an address translation of the base address, it must be guaranteed that it will be used: inserting translation of an address that would not be done otherwise would be a change in program behaviour. To ensure this, the optimisation makes sure that at least one of the users of the base address post-dominates the instruction defining the base address.

The algorithm for combining translation of close addresses is shown in Algorithm 5.5. The code is shown in B.6. Note that the optimisation is iterated as long as it has any effect on the code. MMU emulation calls that are replaced with loads in one iteration may in turn be used as base addresses in a later iteration. The extra iterations add little compilation time.

The effect of running the optimisation on the `foo` function above is shown in Listing 5.8. Note how a single address translation has replaced the two previous calls for MMU emulation.

Table 5.7 shows benchmark performance when run with this optimisation. The α benchmarks receive the highest speed-up, as they apparently have the most occurrences of the pattern targeted by this optimisation.

These three optimisations significantly reduce the number of calls for MMU emulation. Table 5.8 shows the exact numbers for the benchmarks.

for each MMU emulation call I_C in proper function F **do**
  $A \leftarrow$ the address operand of I_C
  if $A =$ `add` I_L, C where I_L is a load and C a constant **then**
    if $C \leq threshold$ **then**
      $users(I_L) \leftarrow uses(I_L) \cup (I_C, C)$
    end if
  else if A is a load instruction **then**
    $users(A) \leftarrow users(A) \cup (I_C, 0)$
  end if
end for
for each base address B in $users$ **do**
  $\mathcal{S} \leftarrow users(B)$
  if $|\mathcal{S}| = 1$ **then**
    continue
  end if
  if There is no $(I_C, C) \in \mathcal{S}$ such that I_C post-dominates B **then**
    continue
  end if
  Normalise $\mathcal{S}$ so that $\exists (I_C, C) \in \mathcal{S}$ such that $C = 0$
  Insert translation of B to host address
  for each $(I_C, C) \in \mathcal{S}$ **do**
    Replace I_C by using translation of B and offset C
  end for
end for

Algorithm 5.5: Combining translation of close addresses.

Benchmark	Execution (s)		Compile (s)		Translations		Speed-up
	mean	σ	mean	σ	Replaced	New	
A	204.82	0.16	14.38	0.10	927	190	1.20
B	103.31	0.09	14.39	0.05	927	190	1.20
C	115.96	0.07	12.91	0.13	255	70	1.62
D	206.86	0.07	11.63	0.07	255	70	1.62

Table 5.7: Performance with combined translation of close addresses.

Benchmark	`emul_read32`	`emul_write32`	`emul_translate_addr`
A	1,431	1,326	953
B	1,431	1,326	953
C	285	285	1,157
D	259	258	1,010

Table 5.8: MMU calls in benchmarks after optimisations.

```
define void @f(%0* noalias) {
L0:
  %p = getelementptr inbounds %0* %0, i32 0, i32 1, i32 3
  br label %L1

L1:
  %c = phi i32 [ 100, %L0 ], [ %d, %L1 ]
  %a = load i32* %p
  %b = add i32 %a, 1
  store i32 %b, i32* %p
  %d = sub i32 %c, 1
  %x = icmp eq i32 %d, 0
  br i1 %x, label %L2, label %L1

L2:
  ret void
}
```

Listing 5.9: Loop with accesses to loop-invariant addresses before LICM.

```
define void @f(%0* noalias) {
L0:
  %p = getelementptr inbounds %0* %0, i32 0, i32 1, i32 3
  %p.promoted = load i32* %p
  br label %L1

L1:
  %p.tmp.0 = phi i32 [ %p.promoted, %L0 ], [ %b, %L1 ]
  %c = phi i32 [ 100, %L0 ], [ %d, %L1 ]
  %b = add i32 %p.tmp.0, 1
  %d = sub i32 %c, 1
  %x = icmp eq i32 %d, 0
  br i1 %x, label %L2, label %L1

L2:
  store i32 %b, i32* %p
  ret void
}
```

Listing 5.10: Loop with accesses to loop-invariant addresses after LICM.

5.3 Loop-Invariant Code Motion

Loop-invariant code motion (LICM), sometimes called hoisting, is the process of moving code that does not need to be in a loop out of it. For example, a store instruction that writes to the same memory location (such as a register variable in the state object), can be moved out of the loop because only the last store is important. Similarly, a load instruction that reads from the same memory location each loop iteration can be moved out. These loads and stores are not dead, so they are not removed by dead load and store elimination. However, they are frequently occurring, and moving them out of loops makes the loops run faster.

This optimisation is already implemented in LLVM and thus easily turned on in the emulator. The effects on code can be seen by comparing Listing 5.9 (before LICM), and Listing 5.10.

The effect of LICM on the benchmarks is shown in Table 5.9. It benefits the β benchmarks and does not affect the α benchmarks much (it actually slows them down a little for some reason). It is cheap in terms of compilation time, and because of its benefit on the β benchmarks, it is kept.

Benchmark	Execution (s)		Compile (s)		Speed-up
	mean	σ	mean	σ	
A	206.12	0.10	14.28	0.07	1.19
B	103.92	0.09	14.25	0.09	1.19
C	111.59	0.07	12.84	0.08	1.69
D	199.28	0.12	11.56	0.05	1.68

Table 5.9: Benchmarks after loop-invariant code motion.

5.4 Changing Calling Convention

A calling convention is a scheme for generating machine code that implements function calls and returns. There are many ways to do this, and they differ between hardware platforms and operating systems.

On Intel x86 processors, calling a function normally involves pushing the function arguments on the stack, making the actual call (storing return address on stack and setting program counter to the function's location), and looking for the return value in the `eax` register afterwards.

Here is an example from the native code generated by the JIT compiler:

```
0xb695f721: mov    DWORD PTR [esp],esi
0xb695f724: call   0xb69ea040
0xb695f729: test   eax,eax
0xb695f72b: jne    0xb695f770
```

As described in Section 4.3.2, function objects take a pointer to the state variable as argument. In this example, that pointer is stored in the `esi` register, and is moved to the stack which is pointed to by `esp`. Room has been made on the stack earlier. After the call, a test is made to see if the return value, in register `eax`, is non-zero.

LLVM allows for the use of different calling conventions. It provides a fast calling convention which passes arguments in registers instead of on the stack, thereby reducing memory accesses. As this does not follow the standard convention, it can only be used internally: in functions that are defined within the emulator and only called internally.

When using the fast calling convention, the example looks like this:

```
0xb695e5ba: call   0xb69e9048
0xb695e5bf: test   eax,eax
0xb695e5c1: jne    0xb695e67c
```

The argument to the function was placed in `ecx` earlier, which is where the called function looks for it. This is a very small optimisation. The code generated with the default convention is already quite good, but this gives an opportunity for the code generator to do an even better job.

Benchmark performance when running with this optimisation together with the previous ones is shown in Table 5.10. The effect on the benchmarks is so tiny that this optimisation cannot be considered valuable, and it is therefore not used further.

Benchmark	Execution (s)		Compile (s)		Speed-up
	mean	σ	mean	σ	
A	206.22	0.31	14.30	0.07	1.19
B	103.89	0.11	14.29	0.09	1.19
C	111.59	0.08	12.83	0.06	1.69
D	199.14	0.10	11.61	0.04	1.68

Table 5.10: Benchmarks with fast calling convention.

5.5 Nice Functions

Recall from Section 4.3.2 that generated functions return the value 1 when they want to get out of emulation, and subsequently that all function calls must be followed by checks of the return value. In many cases, these checks are unnecessary as the function is *nice* and can never return 1. The checks imply instructions for testing and branching, which can be removed if the call is to a nice function.

This optimisation investigates whether a function is nice. If a function is nice, calls of the function can be simplified: the return value check can be removed. When a call has been simplified it is possible that the entire calling function becomes nice, and so the niceness propagates in the call tree.

The optimisation is implemented in two places:

1. During function building, when making a call, the called function is looked up in a table of nice functions. If the callee is in the table, no return value check is inserted.

2. After a function is built, it is *nicified*. This means going through the function, looking to see if a non-zero value is ever returned. If it is not, the function is registered in the table of nice functions, calls to it are simplified, and affected calling functions are nicified.

Benchmark performance when doing nice function optimisation together with previous optimisations is shown in Table 5.11. As can be seen, the performance is not affected much (the effect is even slightly negative). Simplifying the call sites may seem like a nice idea in theory, but it does not seem to have any practical effect on these benchmarks. This optimisation is not used further.

Benchmark	Execution (s)		Compile (s)		Functions		Speed-up
	mean	σ	mean	σ	Nice	Total	
A	208.04	0.15	14.62	0.10	125	259	1.18
B	104.85	0.12	14.68	0.12	125	259	1.18
C	111.84	0.18	13.09	0.07	67	162	1.68
D	200.39	0.35	11.76	0.07	66	158	1.67

Table 5.11: Benchmarks with nice function optimisation.

5.6 Function Inlining

Function inlining is the process of replacing function call instructions with the bodies of the called functions. This has two main advantages:

1. The removal of the call overhead. There is some work involved in making a call, and it hurts instruction cache locality. If the called function is very small, the overhead of calling it may be large compared to the execution time of the function body.

2. Increased scope for optimisations. For example, if a function is called inside a loop, inlining it may expose some loop-invariant variables which can be moved out of the loop with LICM. Other optimisations, such as dead load and store elimination, the MMU emulation optimisations, etc., all benefit: the wider their scope, the more they can do.

The main disadvantage of function inlining is code growth. Compiling more code takes more time: both for optimisations and code generation. Executing more code may also be slower, because of increased pressure on the processor's instruction cache. The inliner must take this into consideration when making decisions about where to inline a function.

LLVM provides a pass for function inlining. The pass works on an entire module and performs inlining where it sees fit. However, it is not suitable for use in the emulator as the module there is built gradually: all functions are not built at once, so inlining needs to be performed gradually, once every time a new function is built. The utility functions that perform the inlining at a call site can be reused, however.

Implementation of a custom inlining pass, suitable for running after a function is built, is shown in B.7. The pass first looks at all function calls made within the function and decide if they should be inlined. Then, it looks at all callers of the just built function, and decides if it should be inlined into them.

Three criteria are used to decide where to perform inline expansion of a function call:

1. Callee size: if the called function is small, the cost of inlining it is small.

2. Leaf functions: if the called function is a leaf in the call graph (it does not itself make any function calls), then it should have priority.

3. Function calls in loops: if a call is made in a loop, it is probably made several times. Valuable optimisations such as LICM and moving address translation out of loops may be made possible if the call is inlined.

When a newly built function is inlined into an already built and optimised function, that function will be re-optimised. This adds some compilation time, but not very much.

Function inlining often involves deleting the called function if all calls to it are inlined. In the emulator, however, there is no way to know if a function may be called in the future. Therefore, functions are never deleted by the inliner.

Benchmark performance and number of inlined functions is shown in Table 5.12. As can be seen, the β benchmarks receive a small speed-up, and the α benchmarks are not much affected. One might have hoped for a larger speed-up, but at least the operation is not too expensive in terms of increased compilation time.

Benchmark	Execution (s)		Compile (s)		Inlined	Speed-up
	mean	σ	mean	σ		
A	207.99	0.12	15.54	0.10	94	1.18
B	104.78	0.09	15.54	0.09	94	1.18
C	111.05	0.06	13.87	0.05	53	1.69
D	198.77	0.27	12.49	0.07	53	1.69

Table 5.12: Benchmarks with function inlining.

5.7 Loop Unrolling

Loop unrolling is the process of transforming a loop, which executes the same code a number of iterations, into continuous code. One can think of it as a number of the loop's bodies being copied and laid out after each other. The main idea is to give optimisation passes a wider scope to work with. LLVM provides a pass to perform loop unrolling, ready to be used.

The effect of using loop unrolling together with previous optimisations on the benchmarks is shown in Table 5.13. This optimisation provides a small speed-up for the α benchmarks, but does not affect the β benchmarks very much, though there are several unrolled loops. The small gain makes it doubtable whether this optimisation is worth its cost.

Benchmark	Execution (s)		Compile (s)		Unrolled		Speed-up
	mean	σ	mean	σ	Partly	Fully	
A	206.84	0.11	15.18	0.04	14	21	1.18
B	104.51	0.07	15.15	0.08	14	21	1.18
C	110.98	0.09	14.00	0.07	53	45	1.69
D	198.63	0.06	12.65	0.03	47	37	1.69

Table 5.13: Benchmarks with function loop unrolling.

5.8 Register Allocation and Code Generation

LLVM provides a couple of different register allocation implementations. The one used by the emulator is the default allocator known as the local register allocator. There is also a linear scan register allocator, which is supposedly faster, and the partitioned boolean quadratic programming (PBQP) register allocator, which is more heavyweight. None of these make any significant difference to the emulator's performance, so the default is determined to be a good choice.

The code generator, that is the process that finally translates LLVM IR to machine code, has different optimisation levels. The emulator has been using the default level, but there are also levels for none, less, or aggressive optimisations. Going for the aggressive optimisations does not make the code emulator go any faster. Using the low-effort levels "none" and "less" makes compilation faster,

but not as much as it makes execution slower, so they are no good either. The emulator continues to use the default level.

Chapter 6

Conclusions

Figure 6.1 illustrates the benchmarks' performance before and after optimisation. As can be seen, there was some performance to be gained by applying compiler optimisations. Especially the β benchmarks benefit, as their execution is dominated by RASC emulation, with the D benchmark's running time going from more than five minutes to less than four. As illustrated by the figure, the increase in compilation time is small compared to the decrease in execution time.

One might have hoped for even better speed-ups as not many optimisations were turned on in the baseline. But one must remember that the firmware is already optimised once. The optimisations that prove to be most useful, that is dead load and store elimination and reduction of MMU emulation, are mostly cleaning up redundancy introduced by the JIT-compiler. Trying to perform optimisations such as function inlining and loop unrolling, that are typically not done on the RASC firmware because of the target's small caches, has not really been successful. There may still be some performance to gain there, but it requires more careful implementation and tuning.

Some optimisations are not beneficial for all benchmarks. This thesis proposes that the optimisations be added to the emulator as options that can be activated by command-line switches. Optimisations such as dead load and store elimination and MMU emulation reduction should go in at a default optimisation level, whereas loop-invariant code motion, inlining and unrolling should be left as separate flags which can be turned on by the expert user.

All in all, an execution time speed-up of about 70 % for RASC-dominated benchmarks, and 20 % for the others should make the users of the emulator happy.

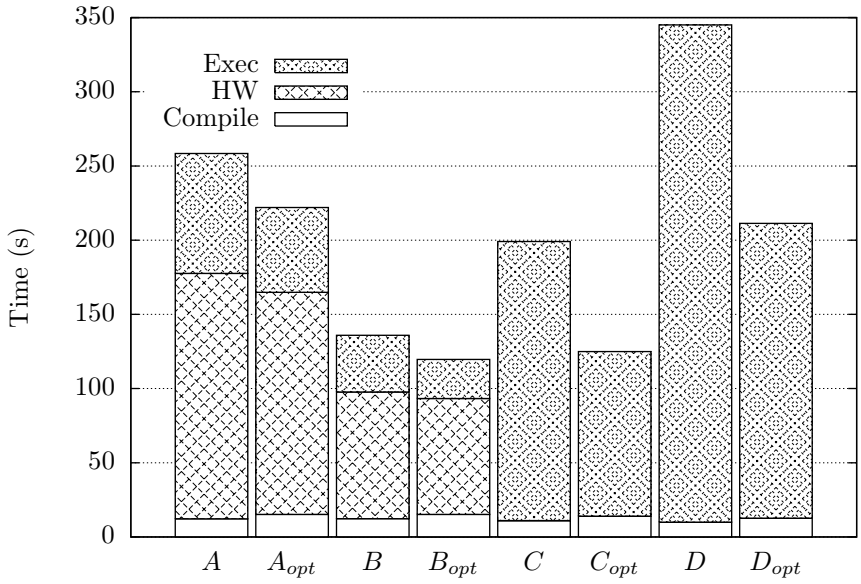

Figure 6.1: Benchmarks before and after optimisation.

Appendix A

Performance Measurements

The following tables present measurements of execution and compilation time for the benchmarks using different optimisations. Each benchmark has been executed ten times, and the means and standard deviations are presented. The speed-up column refers to speed-up of execution time.

§ 5.1.2	§ 5.1.3	§ 5.2.1	§ 5.2.2	§ 5.2.3	§ 5.3	§ 5.4	§ 5.5	§ 5.6	§ 5.7	Exec (s)		Compile (s)		Instr	Speed-up
										mean	σ	mean	σ		
										246.13	0.41	12.25	0.06	78,896	1.00
•										244.88	0.46	13.72	0.08	60,222	1.01
•	•									244.14	0.22	14.60	0.09	57,877	1.01
•	•	•								240.02	0.21	14.34	0.07	57,489	1.03
•	•	•	•							232.98	0.12	14.67	0.11	58,339	1.06
•	•	•	•	•						204.82	0.16	14.38	0.10	59,476	1.20
•	•	•	•	•	•					206.12	0.10	14.28	0.07	59,611	1.19
•	•	•	•	•	•	•				206.22	0.31	14.30	0.07	59,611	1.19
•	•	•	•	•	•		•			208.04	0.15	14.62	0.10	58,752	1.18
•	•	•	•	•	•			•		207.99	0.12	15.54	0.10	61,256	1.18
•	•	•	•	•	•			•	•	206.84	0.11	15.18	0.04	61,899	1.19

Table A.1: Benchmark A executed 10 times with different optimisations: global value numbering (§ 5.1.2), improved dead-store elimination (§ 5.1.3), compile-time constant address translation (§ 5.2.1), moving address translation out of loops (§ 5.2.2), combining translation of close addresses (§ 5.2.3), loop-invariant code motion (§ 5.3), fast calling convention (§ 5.4), nice functions (§ 5.5), function inlining (§ 5.6) and loop unrolling (§ 5.7) .

§ 5.1.2	§ 5.1.3	§ 5.2.1	§ 5.2.2	§ 5.2.3	§ 5.3	§ 5.4	§ 5.5	§ 5.6	§ 5.7	Exec (s)		Compile (s)		Instr	Speed-up
										mean	σ	mean	σ		
										123.66	0.17	12.28	0.08	78,896	1.00
•										123.43	0.10	13.75	0.07	60,222	1.00
•	•									123.37	0.27	14.59	0.11	57,877	1.00
•	•	•								121.17	0.18	14.37	0.09	57,489	1.02
•	•	•	•							117.79	0.20	14.67	0.09	58,339	1.05
•	•	•	•	•						103.31	0.09	14.39	0.05	59,476	1.20
•	•	•	•	•	•					103.92	0.09	14.25	0.09	59,611	1.19
•	•	•	•	•	•	•				103.89	0.11	14.29	0.09	59,611	1.19
•	•	•	•	•	•		•			104.85	0.12	14.68	0.12	58,752	1.18
•	•	•	•	•	•			•		104.78	0.09	15.54	0.09	61,256	1.18
•	•	•	•	•	•			•	•	104.51	0.07	15.15	0.08	61,900	1.18

Table A.2: Benchmark B executed 10 times with different optimisations: global value numbering (§ 5.1.2), improved dead-store elimination (§ 5.1.3), compile-time constant address translation (§ 5.2.1), moving address translation out of loops (§ 5.2.2), combining translation of close addresses (§ 5.2.3), loop-invariant code motion (§ 5.3), fast calling convention (§ 5.4), nice functions (§ 5.5), function inlining (§ 5.6) and loop unrolling (§ 5.7) .

§ 5.1.2	§ 5.1.3	§ 5.2.1	§ 5.2.2	§ 5.2.3	§ 5.3	§ 5.4	§ 5.5	§ 5.6	§ 5.7	Exec (s)		Compile (s)		Instr	Speed-up
										mean	σ	mean	σ		
										188.06	0.11	11.07	0.08	63,001	1.00
•										181.33	0.08	12.45	0.04	48,974	1.04
•	•									171.88	0.08	13.17	0.07	46,651	1.09
•	•	•								133.17	0.26	12.62	0.07	45,719	1.41
•	•	•	•							116.32	0.09	12.94	0.06	46,613	1.62
•	•	•	•	•						115.96	0.07	12.91	0.13	46,872	1.62
•	•	•	•	•	•					111.59	0.07	12.84	0.08	46,784	1.69
•	•	•	•	•	•	•				111.59	0.08	12.83	0.06	46,784	1.69
•	•	•	•	•	•		•			111.84	0.18	13.09	0.07	46,454	1.68
•	•	•	•	•	•			•		111.05	0.06	13.87	0.05	47,748	1.69
•	•	•	•	•	•			•	•	110.98	0.09	14.00	0.07	49,413	1.69

Table A.3: Benchmark C executed 10 times with different optimisations: global value numbering (§ 5.1.2), improved dead-store elimination (§ 5.1.3), compile-time constant address translation (§ 5.2.1), moving address translation out of loops (§ 5.2.2), combining translation of close addresses (§ 5.2.3), loop-invariant code motion (§ 5.3), fast calling convention (§ 5.4), nice functions (§ 5.5), function inlining (§ 5.6) and loop unrolling (§ 5.7) .

§ 5.1.2	§ 5.1.3	§ 5.2.1	§ 5.2.2	§ 5.2.3	§ 5.3	§ 5.4	§ 5.5	§ 5.6	§ 5.7	Exec (s)		Compile (s)		Instr	Speed-up
										mean	σ	mean	σ		
										335.11	0.13	9.99	0.06	57,033	1.00
•										321.24	0.09	11.17	0.05	44,430	1.04
•	•									302.47	0.11	11.84	0.06	42,362	1.11
•	•	•								233.70	0.39	11.38	0.09	41,581	1.43
•	•	•	•							207.57	0.09	11.70	0.05	42,341	1.61
•	•	•	•	•						206.86	0.07	11.63	0.07	42,600	1.62
•	•	•	•	•	•					199.28	0.12	11.56	0.05	42,530	1.68
•	•	•	•	•	•	•				199.14	0.10	11.61	0.04	42,530	1.68
•	•	•	•	•	•		•			200.39	0.35	11.76	0.07	42,200	1.67
•	•	•	•	•	•			•		198.77	0.27	12.49	0.07	43,494	1.69
•	•	•	•	•	•			•	•	198.63	0.06	12.65	0.03	44,991	1.69

Table A.4: Benchmark D executed 10 times with different optimisations: global value numbering (§ 5.1.2), improved dead-store elimination (§ 5.1.3), compile-time constant address translation (§ 5.2.1), moving address translation out of loops (§ 5.2.2), combining translation of close addresses (§ 5.2.3), loop-invariant code motion (§ 5.3), fast calling convention (§ 5.4), nice functions (§ 5.5), function inlining (§ 5.6) and loop unrolling (§ 5.7) .

Appendix B

Code Listings

B.1 Null Pointers Do Not Alias

```
--- llvm/trunk/lib/Analysis/BasicAliasAnalysis.cpp (original)
+++ llvm/trunk/lib/Analysis/BasicAliasAnalysis.cpp Mon Nov  9 13:29:11 2009
@@ -646,6 +646,15 @@
   const Value *O1 = V1->getUnderlyingObject();
   const Value *O2 = V2->getUnderlyingObject();

+  // Null values in the default address space don't point to any object, they
+  // don't alias any other pointer.
+  if (const ConstantPointerNull *CPN = dyn_cast<ConstantPointerNull>(O1))
+    if (CPN->getType()->getAddressSpace() == 0)
+      return NoAlias;
+  if (const ConstantPointerNull *CPN = dyn_cast<ConstantPointerNull>(O2))
+    if (CPN->getType()->getAddressSpace() == 0)
+      return NoAlias;
+
   if (O1 != O2) {
     // If V1/V2 point to two different objects we know that we have no alias.
     if (isIdentifiedObject(O1) && isIdentifiedObject(O2))
```

B.2 Constant Pointers and Aliasing

```
--- llvm/trunk/lib/Analysis/BasicAliasAnalysis.cpp (original)
+++ llvm/trunk/lib/Analysis/BasicAliasAnalysis.cpp Sat Nov 14 00:15:14 2009
@@ -659,7 +659,12 @@
     // If V1/V2 point to two different objects we know that we have no alias.
     if (isIdentifiedObject(O1) && isIdentifiedObject(O2))
       return NoAlias;
-
+
+    // Constant pointers can't alias non-const isIdentifiedObject objects.
+    if ((isa<Constant>(O1) && isIdentifiedObject(O2) && !isa<Constant>(O2))||
+        (isa<Constant>(O2) && isIdentifiedObject(O1) && !isa<Constant>(O1)))
+      return NoAlias;
+
     // Arguments can't alias with local allocations or noalias calls.
     if ((isa<Argument>(O1) && (isa<AllocaInst>(O2) || isNoAliasCall(O2))) ||
         (isa<Argument>(O2) && (isa<AllocaInst>(O1) || isNoAliasCall(O1))))
```

B.3 Number of Uses of Store Instructions

```
void DSE::checkStoreUses(Function &F) {
  MemoryDependenceAnalysis &MD = getAnalysis<MemoryDependenceAnalysis>();

  for (Function::iterator FI = F.begin(), FE = F.end(); FI != FE; ++FI) {
    for (BasicBlock::iterator BI = FI->begin(), BE = FI->end();
        BI != BE; ++BI) {
      Instruction *Inst = BI;

      if (isa<LoadInst>(Inst) || isa<CallInst>(Inst)) {
        // Potential user of stores
        MemDepResult InstDep = MD.getDependency(Inst);

        if (InstDep.isNonLocal()) {
          SmallVector<MemoryDependenceAnalysis::NonLocalDepEntry, 64> Deps;

          if (isa<LoadInst>(Inst)) {
            Value *Addr = Inst->getOperand(0);
            MD.getNonLocalPointerDependency(Addr, true, FI, Deps);
          } else if (CallInst *CI = dyn_cast<CallInst>(Inst)) {
            CallSite CS = CallSite::get(CI);
            for (unsigned i = 0, e = CS.arg_size(); i != e; ++i) {
              Value *V = CS.getArgument(i);

              if (isa<PointerType>(V->getType())) {
                SmallVector<MemoryDependenceAnalysis::NonLocalDepEntry,64> R;
                MD.getNonLocalPointerDependency(V, true, FI, R);
                Deps.append(R.begin(), R.end());
              }
            }
          }

          for (unsigned i = 0, e = Deps.size(); i != e; ++i) {
            MemDepResult &DepInfo = Deps[i].second;
            Instruction *I = DepInfo.getInst();

            if (isa<StoreInst>(I))
              ++nbrStoreUses[I];
          }
        } else {
          // Local dependency
          Instruction *I = InstDep.getInst();
          if (isa<StoreInst>(I))
            ++nbrStoreUses[I];
        }
      }
    }
  }
}
```

B.4 DSE With Non-Local Dependencies

```
    if (InstDep.isNonLocal()) {
      StoreInst *SI = dyn_cast<StoreInst>(Inst);
      if (!SI)
        continue;

      SmallVector<MemoryDependenceAnalysis::NonLocalDepEntry, 64> Deps;
      MD.getNonLocalPointerDependency(Inst->getOperand(1), false, &BB, Deps);

      if (Deps.size() == 1 && Deps[0].second.isClobber())
        continue; // Phi translation failure

      if (Deps.size() == 1 && Deps[0].second.isDef()) {
        MemDepResult& DepInfo = Deps[0].second;

        if (LoadInst *DepLoad = dyn_cast<LoadInst>(DepInfo.getInst())) {
          if (SI->getPointerOperand() == DepLoad->getPointerOperand() &&
```

```
            SI->getOperand(0) == DepLoad) {
          WeakVH NextInst(BBI);
          DeleteDeadInstruction(SI);

          if (NextInst == 0)
            BBI = BB.begin();
          else if (BBI != BB.begin())
            --BBI;
          NumFastStores++;
          MadeChange = true;
          continue;
        }
      }
    }

    for (unsigned i = 0, e = Deps.size(); i != e; ++i) {
      BasicBlock *DepBB = Deps[i].first;
      MemDepResult& DepInfo = Deps[i].second;

      if (DepInfo.isClobber())
        continue;

      if (!DepInfo.isDef())
        continue;

      Instruction *DepInst = DepInfo.getInst();
      if (StoreInst *DepStore = dyn_cast<StoreInst>(DepInst)) {
        if (nbrStoreUses[DepStore] != 0)
          continue;

        if (DepStore->isVolatile())
          continue;

        if (PDT.properlyDominates(&BB, DepBB)) {
          if (isStoreAtLeastAsWideAs(Inst, DepStore, TD)) {
            // DepStore is a dead store
            DeleteDeadInstruction(DepStore);
            NumFastStores++;
            MadeChange = true;

            BBI = Inst;
            if (BBI != BB.begin())
              --BBI;
          }
        }
      }
    }
    continue;
  }
```

B.5 Moving Address Translation Out of Loops

```
#define DEBUG_TYPE "MemLoop"

#include "llvm/Pass.h"
#include "llvm/Analysis/LoopPass.h"
#include "llvm/Analysis/ScalarEvolution.h"
#include "llvm/Analysis/ScalarEvolutionExpander.h"
#include "llvm/Analysis/ScalarEvolutionExpressions.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/Support/CallSite.h"
#include "llvm/Support/IRBuilder.h"
#include "llvm/Analysis/PostDominators.h"

#include "rasc.h"

using namespace llvm;

namespace {
```

```
    class MemLoopPass : public LoopPass {
    public:
        static char ID;

        MemLoopPass();
        bool runOnLoop(Loop* L, LPPassManager &LPM);

    private:
        ScalarEvolution *SE;
        DominatorTree *DT;
        PostDominatorTree *PDT;
        Function *translateIfLoop;

        virtual void getAnalysisUsage(AnalysisUsage& AU) const {
            AU.addRequired<ScalarEvolution>();
            AU.addRequired<DominatorTree>();
            AU.addRequired<PostDominatorTree>();
        }

        bool processCall(Loop *L, CallInst *CI);

        void replaceTranslate(Loop *L, Value *TC, CallInst *CI, Value *Addr,
                Value *Base);

        void replaceRead(Loop *L, Value *TC, CallInst *CI, Value *Addr,
                Value *Base);

        void replaceWrite(Loop *L, Value *TC, CallInst *CI, Value *Addr,
                Value *Base, Value *WriteVal);

        Value *newAddrTranslation(Loop *L, Value *TC, CallInst *CI,
                Value *Addr, Value *Base);

        void findInsertPoint(BasicBlock *&Block, BasicBlock::iterator &Pos,
                Value *TC, Value *Base, Loop *L);
    };
}

LoopPass *createMemLoopPass() {
    return new MemLoopPass();
}


char MemLoopPass::ID = 0;
static RegisterPass<MemLoopPass>
X("MemLoopPass", "Move address translation out of loops");

STATISTIC(NumMemLoop, "Addr translations moved out of loops");
STATISTIC(NumTranslate, "emul_translate_addr moved out of loops");
STATISTIC(NumTranslateIfLoop, "emul_translate_addr_if_loop moved");
STATISTIC(NumRead, "emul_read32 moved out of loops");
STATISTIC(NumWrite, "emul_write32 moved out of loops");


// Convenience variables
namespace {
    const Type *int8Ty = Type::getInt8Ty(getGlobalContext());
    const Type *int32Ty = Type::getInt32Ty(getGlobalContext());
    const PointerType *VoidPTy = PointerType::getUnqual(int8Ty);
    Value *const_0 = llvm::ConstantInt::get(int32Ty, 0);
    Value *const_1 = llvm::ConstantInt::get(int32Ty, 1);
    Value *const_4 = llvm::ConstantInt::get(int32Ty, 4);
}

// External variables that we need
extern Function *emul_translate_addrV;
extern Module *TheModule;

/*
   Create function that performs address translation when the
   loop count is != 0, and returns garbage otherwise (garbage
   which should not be used, as the loop should never run).
 */
static Function *createTranslateIfLoop() {
```

```
    std::vector<const llvm::Type*> tmp_args;
    tmp_args.push_back(int32Ty); // addr
    tmp_args.push_back(int32Ty); // trip count

    FunctionType *fty = FunctionType::get(VoidPTy, tmp_args, false);

    Function *f = Function::Create(fty, Function::InternalLinkage,
            "emul_translate_addr_if_loop", TheModule);

    Function::arg_iterator args = f->arg_begin();
    Argument *Addr = args++;
    Argument *Count = args++;

    BasicBlock *bbStart = BasicBlock::Create(getGlobalContext(), "", f);
    BasicBlock *bbZero = BasicBlock::Create(getGlobalContext(), "", f);
    BasicBlock *bbNotZero = BasicBlock::Create(getGlobalContext(), "", f);

    IRBuilder<false> Builder(getGlobalContext());

    Builder.SetInsertPoint(bbStart);
    Value *Test = Builder.CreateICmpEQ(Count, const_0);
    Builder.CreateCondBr(Test, bbZero, bbNotZero);

    Builder.SetInsertPoint(bbZero);
    Builder.CreateRet(Builder.CreateIntToPtr(
                ConstantInt::get(int32Ty, 0xdeadbeef), VoidPTy));

    Builder.SetInsertPoint(bbNotZero);
    Value *Res = Builder.CreateCall3(
            emul_translate_addrV,
            Addr,
            const_4,
            ConstantPointerNull::get(VoidPTy));
    Builder.CreateRet(Res);

    return f;
}

MemLoopPass::MemLoopPass() : LoopPass(&ID) {
    translateIfLoop = createTranslateIfLoop();
}

bool MemLoopPass::runOnLoop(Loop *L, LPPassManager &LPM) {
    bool Changed = false;

    SE = &getAnalysis<ScalarEvolution>();
    DT = &getAnalysis<DominatorTree>();
    PDT = &getAnalysis<PostDominatorTree>();

    for (Loop::block_iterator LI = L->block_begin(), LE = L->block_end();
            LI != LE; ++LI) {

        BasicBlock *BB = *LI;

        for (BasicBlock::iterator BI = BB->begin(), BE = BB->end(), BINext;
                BI != BE; BI = BINext) {

            /* Remember next instr, we might erase the current one */
            BINext = BI; ++BINext;

            if (CallInst *CI = dyn_cast<CallInst>(BI)) {
                Function *Callee = CI->getCalledFunction();

                if (Callee->getName().front() != 'e')
                    continue;

                Changed |= processCall(L, CI);
            }
        }
    }

    return Changed;
}
```

```
bool MemLoopPass::processCall(Loop *L, CallInst *CI) {
    enum {
        TRANS,
        TRANS_IF_LOOP,
        READ,
        WRITE
    } Type;

    Function *Callee = CI->getCalledFunction();
    const StringRef &FuncName = Callee->getName();

    if (FuncName.equals("emul_translate_addr"))
        Type = TRANS;
    else if (FuncName.equals("emul_translate_addr_if_loop"))
        Type = TRANS_IF_LOOP;
    else if (FuncName.equals("emul_read32"))
        Type = READ;
    else if (FuncName.equals("emul_write32"))
        Type = WRITE;
    else
        return false;

    // Call has to post-dominate loop header.
    if (!PDT->dominates(CI->getParent(), L->getHeader()))
        return false;

    Value *TC = L->getTripCount();
    if (TC == NULL) {
        // We have do be able to determine trip count to move anything safely
        return false;
    }

    CallSite CS = CallSite::get(CI);
    Value *Addr = CS.getArgument(0); /* Addr to operate on is always arg 0 */

    const SCEV *AddrSCEV = SE->getSCEV(Addr);
    if (const SCEVAddRecExpr *ARE = dyn_cast<SCEVAddRecExpr>(AddrSCEV)) {
        /* Our analysis is based on the Addr value being on this form */

        const SCEV *Start = ARE->getStart();

        SCEVExpander Expander(*SE);
        Value *Base = Expander.expandCodeFor(Start, int32Ty,
                L->getHeader()->begin());

        switch(Type) {
            case TRANS:
                replaceTranslate(L, TC, CI, Addr, Base);
                ++NumTranslate;
                ++NumMemLoop;
                return true;
            case TRANS_IF_LOOP:
                /* We can only translate if the OldTC is known not zero */
                if (const SCEV *OldTC = SE->getSCEV(CS.getArgument(1))) {
                    if (SE->isKnownNonZero(OldTC)) {
                        replaceTranslate(L, TC, CI, Addr, Base);
                        ++NumTranslateIfLoop;
                        ++NumMemLoop;
                        return true;
                    }
                }
                return false;
            case READ:
                replaceRead(L, TC, CI, Addr, Base);
                ++NumRead;
                ++NumMemLoop;
                return true;
            case WRITE:
                replaceWrite(L, TC, CI, Addr, Base, CS.getArgument(1));
                ++NumWrite;
                ++NumMemLoop;
                return true;
            default:
                assert(0 && "Unreachable switch case reached");
```

```
                break;
        }

    }

    return false;
}

void MemLoopPass::replaceTranslate(Loop *L, Value *TC, CallInst *CI,
        Value *Addr, Value *Base) {

    Value *A = newAddrTranslation(L, TC, CI, Addr, Base);

    CI->replaceAllUsesWith(A);
    CI->eraseFromParent();
}

void MemLoopPass::replaceRead(Loop *L, Value *TC, CallInst *CI,
        Value *Addr, Value *Base) {

    Value *A = newAddrTranslation(L, TC, CI, Addr, Base);

    IRBuilder<false> Builder(getGlobalContext());
    Builder.SetInsertPoint(CI->getParent(), CI);

    Value *AA = Builder.CreatePointerCast(A,PointerType::getUnqual(int32Ty));
    Value *Load = Builder.CreateLoad(AA);

    CI->replaceAllUsesWith(Load);
    CI->eraseFromParent();
}

void MemLoopPass::replaceWrite(Loop *L, Value *TC, CallInst *CI,
        Value *Addr, Value *Base, Value *WriteVal) {

    Value *A = newAddrTranslation(L, TC, CI, Addr, Base);

    IRBuilder<false> Builder(getGlobalContext());
    Builder.SetInsertPoint(CI->getParent(), CI);

    Value *AA = Builder.CreatePointerCast(A,PointerType::getUnqual(int32Ty));
    Builder.CreateStore(WriteVal, AA);

    CI->eraseFromParent();
}


Value *MemLoopPass::newAddrTranslation(Loop *L, Value *TC, CallInst *CI,
        Value *Addr, Value *Base)
{
    IRBuilder<false> Builder(getGlobalContext());

    BasicBlock *TargetBlock;
    BasicBlock::iterator TargetPos;

    findInsertPoint(TargetBlock, TargetPos, TC, Base, L);
    Builder.SetInsertPoint(TargetBlock, TargetPos);

    Value *BasePtr;
    if (isa<ConstantInt>(Base)) {
        ConstantInt *C = dyn_cast<ConstantInt>(Base);

        BasePtr = Builder.CreateIntToPtr(ConstantInt::get(
                    int32Ty,
                    (uint32_t)emul_translate_addr(
                        C->getZExtValue(),
                        4,
                        NULL)),
                VoidPTy);
    } else {
        BasePtr = Builder.CreateCall2(
                translateIfLoop,
                Base,
                Builder.CreateTrunc(TC, int32Ty));
```

```
    }

    Value *BaseVal = Builder.CreatePtrToInt(BasePtr, int32Ty);

    Builder.SetInsertPoint(CI->getParent(), CI);
    Value *Offset = Builder.CreateSub(Addr, Base);
    Value *Sum = Builder.CreateAdd(Offset, BaseVal);
    Value *NewAddr = Builder.CreateIntToPtr(Sum, VoidPTy);

    return NewAddr;
}

/*
   Find an insert point which is dominated by both TC and Base.
 */
void MemLoopPass::findInsertPoint(BasicBlock *&Block,
        BasicBlock::iterator &Pos, Value *TC, Value *Base, Loop *L) {

    Instruction *BaseInst, *TCInst;

    if ((BaseInst = dyn_cast<Instruction>(Base)) &&
            (TCInst = dyn_cast<Instruction>(TC))) {

        if (DT->dominates(BaseInst, TCInst)) {
            // Insert after trip count
            Block = TCInst->getParent();
            Pos = TCInst;
            ++Pos;
        } else if (DT->dominates(TCInst, BaseInst)) {
            // Insert after base
            Block = BaseInst->getParent();
            Pos = BaseInst;
            ++Pos;
        } else if (isa<PHINode>(TCInst) && isa<PHINode>(BaseInst)
                && TCInst->getParent() == BaseInst->getParent()) {
            // Phi nodes don't have dominance relations in the same block.
            // Just pick one as insert point, we'll insert after the phi
            // nodes in the block anyway.
            Block = TCInst->getParent();
            Pos = TCInst;
            ++Pos;
        } else {
            assert(0 && "TCInst and BaseInst not on the same path");
        }
    } else if ((BaseInst = dyn_cast<Instruction>(Base))) {
        // Insert after base
        Block = BaseInst->getParent();
        Pos = BaseInst;
        ++Pos;
    } else if ((TCInst = dyn_cast<Instruction>(TC))) {
        // Insert after trip count
        Block = TCInst->getParent();
        Pos = TCInst;
        ++Pos;
    } else {
        // Just put it somewhere in the loop header
        Block = L->getHeader();
        Pos = Block->begin();
    }

    // Make sure we don't insert among phi nodes
    while (Pos != Block->end() && isa<PHINode>(Pos)) ++Pos;
}
```

B.6 Combining Translation of Close Addresses

```
#define DEBUG_TYPE "MergeAddrTrans"

#include "llvm/Pass.h"
#include "llvm/BasicBlock.h"
```

```
#include "llvm/Instructions.h"
#include "llvm/Support/CallSite.h"
#include "llvm/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/IRBuilder.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/Analysis/PostDominators.h"

#include <vector>
#include <map>
#include <utility>
#include <set>

using namespace llvm;

namespace {
    class MergeCloseAddr : public FunctionPass {
    public:
        static char ID;

        MergeCloseAddr() : FunctionPass(&ID) {};
        bool runOnFunction(Function &F);

    private:
        virtual void getAnalysisUsage(AnalysisUsage& AU) const {
            AU.addRequired<PostDominatorTree>();
            AU.addPreserved<PostDominatorTree>();
        }

        typedef std::vector<std::pair<CallInst*, int> > UserList;
        typedef std::map<Instruction*, UserList> UseMap;

        void processBasicBlock(BasicBlock &BB);
        void processCall(CallInst *CI);
        bool processUserLists();
        bool checkPostDominance(Instruction *Base, UserList& Users);
        Instruction *normaliseOffsets(Instruction *Base, UserList &Users);
        void transformUsers(Instruction *Base, const UserList &users);
        Value* insertTranslation(Instruction *Base, const UserList &users);

        void replaceRead(CallInst *Call, Value *BaseVal, int offset);
        void replaceWrite(CallInst *Call, Value *BaseVal, int offset);
        void replaceTrans(CallInst *Call, Value *BaseVal, int offset);

        void removeInst(Instruction *inst);

        UseMap useMap;

        enum { OFFSET_LIMIT = 512 };
    };
}

FunctionPass *createMergeCloseAddrPass() {
    return new MergeCloseAddr();
}

char MergeCloseAddr::ID = 0;
static RegisterPass<MergeCloseAddr>
X("MergeAddrTrans", "Merge translation of close memory addresses");

STATISTIC(InsertedTranslations, "Inserted translations");
STATISTIC(ReplacedCalls, "Replaced calls");
STATISTIC(NonSecureBases, "Non post-dominated bases");
STATISTIC(Removed, "Removed instructions");

// Convenience variables
namespace {
    const Type *int8Ty = Type::getInt8Ty(getGlobalContext());
    const Type *int32Ty = Type::getInt32Ty(getGlobalContext());
    const PointerType *VoidPTy = PointerType::getUnqual(int8Ty);
    Value *const_0 = ConstantInt::get(int32Ty, 0);
    Value *const_1 = ConstantInt::get(int32Ty, 1);
    Value *const_4 = llvm::ConstantInt::get(int32Ty, 4);
    Value *const_null = ConstantPointerNull::get(VoidPTy);
```

```
}

// External vars that we need
extern Function *emul_translate_addrV;
extern std::set<Function*> proper_functions;

bool MergeCloseAddr::runOnFunction(Function& F) {
    if (proper_functions.find(&F) == proper_functions.end())
        return false;

    bool Changed = false;
    bool LocalChanged;

    do {
        LocalChanged = false;
        useMap.clear();

        for (Function::iterator I = F.begin(), E = F.end(); I != E; ++I)
            processBasicBlock(*I);

        LocalChanged = processUserLists();
        Changed |= LocalChanged;
    } while (LocalChanged);

    return Changed;
}

void MergeCloseAddr::processBasicBlock(BasicBlock &BB) {
    for (BasicBlock::iterator BI=BB.begin(), BE=BB.end(); BI != BE; ++BI) {
        Instruction *I = BI;

        if (CallInst *CI = dyn_cast<CallInst>(I)) {
            Function *Callee = CI->getCalledFunction();
            const StringRef& FuncName = Callee->getName();

            if (FuncName.equals("emul_read32") ||
                    FuncName.equals("emul_write32") ||
                    FuncName.equals("emul_translate_addr")) {

                processCall(CI);
            }
        }
    }
}

void MergeCloseAddr::processCall(CallInst *CI) {
    CallSite CS = CallSite::get(CI);

    Value *Addr = CS.getArgument(0);

    if (BinaryOperator *BinOp = dyn_cast<BinaryOperator>(Addr)) {
        if (BinOp->getOpcode() == Instruction::Add) {
            Value *OpA = BinOp->getOperand(0);
            Value *OpB = BinOp->getOperand(1);

            if (Instruction *I = dyn_cast<Instruction>(OpA)) {
                if (!isa<LoadInst>(I))
                    return;

                if (ConstantInt *C = dyn_cast<ConstantInt>(OpB)) {
                    int offset = C->getSExtValue();
                    Instruction *Base = I;

                    if (offset <= OFFSET_LIMIT)
                        useMap[Base].push_back(std::make_pair(CI, offset));
                }
            }
        }
    } else if (LoadInst *LI = dyn_cast<LoadInst>(Addr)) {
        // If we use a load directly, then offset is zero
        useMap[LI].push_back(std::make_pair(CI, 0));
    }
}
```

```
bool MergeCloseAddr::processUserLists() {
    bool Changed = false;

    for (UseMap::iterator I=useMap.begin(), IE=useMap.end(); I!=IE; ++I) {
        UserList& userList = I->second;
        Instruction *Base = I->first;

        if (userList.size() == 1)
            continue;

        if (checkPostDominance(Base, userList)) {
            Instruction *NewBase = normaliseOffsets(Base, userList);
            transformUsers(NewBase, userList);
            Changed = true;
        }
    }

    return Changed;
}

bool MergeCloseAddr::checkPostDominance(Instruction *Base, UserList& Users) {
    // Check if at least one of the users post-dominates base.

    PostDominatorTree& PDT = getAnalysis<PostDominatorTree>();

    for (size_t i = 0, e = Users.size(); i != e; ++i) {
        Instruction *Inst = Users[i].first;

        BasicBlock *bbInst = Inst->getParent();
        BasicBlock *bbBase = Base->getParent();

        if (bbInst == bbBase)
            return true; // Inst uses Base, so Inst most post-dominate

        if (PDT.properlyDominates(bbInst, bbBase))
            return true;
    }

    ++NonSecureBases;
    return false;
}

Instruction *MergeCloseAddr::normaliseOffsets(Instruction *Base,
                                              UserList &Users) {
    // Normalise so that at least one of the users have 0 offset

    int minOffset = OFFSET_LIMIT + 1;
    for (size_t i = 0, e = Users.size(); i != e; ++i) {
        int offset = Users[i].second;
        assert(offset <= OFFSET_LIMIT);

        minOffset = offset < minOffset ? offset : minOffset;
    }

    if (minOffset == 0)
        return Base;

    int diff = minOffset;
    IRBuilder<false> Builder(getGlobalContext());
    BasicBlock::iterator InsertPoint;
    InsertPoint = Base;
    ++InsertPoint;
    Builder.SetInsertPoint(Base->getParent(), InsertPoint);

    Value *b = Builder.CreateAdd(Base, ConstantInt::get(int32Ty, diff));
    assert(isa<Instruction>(b));

    for (size_t i = 0, e = Users.size(); i != e; ++i)
        Users[i].second -= diff;

    return dyn_cast<Instruction>(b);
}

void MergeCloseAddr::transformUsers(Instruction *Base,
```

```
                                    const UserList &users) {
    // Insert a translation of the base address, and let all users use
    // that together with an offset

    Value *BaseVal = insertTranslation(Base, users);
    ++InsertedTranslations;

    for (size_t i = 0, e = users.size(); i != e; ++i) {
        CallInst *Call = users[i].first;
        int offset = users[i].second;

        const StringRef& FuncName = Call->getCalledFunction()->getName();
        if (FuncName.equals("emul_read32")) {
            replaceRead(Call, BaseVal, offset);
        } else if (FuncName.equals("emul_write32")) {
            replaceWrite(Call, BaseVal, offset);
        } else if (FuncName.equals("emul_translate_addr")) {
            replaceTrans(Call, BaseVal, offset);
        } else {
            assert(0 && "Not reached");
        }

        ++ReplacedCalls;
    }
}

Value* MergeCloseAddr::insertTranslation(Instruction *Base,
                                         const UserList& users) {
    // Insert a translation operation which translates Base
    // address into a host address

    IRBuilder<false> Builder(getGlobalContext());
    BasicBlock::iterator InsertPoint = Base;
    ++InsertPoint;
    Builder.SetInsertPoint(Base->getParent(), InsertPoint);

    Value *p = Builder.CreateCall3(
            emul_translate_addrV,
            Base,
            const_4,
            const_null);

    Value *v = Builder.CreatePtrToInt(p, int32Ty);
    return v;
}

void MergeCloseAddr::replaceRead(CallInst *Call,Value *BaseVal,int offset){
    IRBuilder<false> Builder(getGlobalContext());
    Builder.SetInsertPoint(Call->getParent(), Call);

    Value *t = Builder.CreateAdd(BaseVal, ConstantInt::get(int32Ty, offset));
    Value *t2 = Builder.CreateIntToPtr(t, PointerType::getUnqual(int32Ty));
    Instruction *t3 = Builder.CreateLoad(t2);

    Call->replaceAllUsesWith(t3);
    removeInst(Call);
}

void MergeCloseAddr::replaceWrite(CallInst *Call,Value *BaseVal,int offset){
    IRBuilder<false> Builder(getGlobalContext());
    Builder.SetInsertPoint(Call->getParent(), Call);

    Value *t = Builder.CreateAdd(BaseVal, ConstantInt::get(int32Ty, offset));
    Value *t2 = Builder.CreateIntToPtr(t, PointerType::getUnqual(int32Ty));

    CallSite CS = CallSite::get(Call);
    Builder.CreateStore(CS.getArgument(1), t2);

    removeInst(Call);
}

void MergeCloseAddr::replaceTrans(CallInst *Call,Value *BaseVal,int offset){
    IRBuilder<false> Builder(getGlobalContext());
    Builder.SetInsertPoint(Call->getParent(), Call);
```

```
    Value *t = Builder.CreateAdd(BaseVal, ConstantInt::get(int32Ty, offset));
    Value *t2 = Builder.CreateIntToPtr(t, VoidPTy);

    Call->replaceAllUsesWith(t2);
    removeInst(Call);
}

void MergeCloseAddr::removeInst(Instruction *I) {
    for (unsigned i = 0, e = I->getNumOperands(); i != e; ++i) {
        Value *Op = I->getOperand(i);
        I->setOperand(i, 0);

        if (Op->use_empty())
            if (Instruction *NowDead = dyn_cast<Instruction>(Op))
                removeInst(NowDead);
    }

    I->eraseFromParent();
    ++Removed;
}
```

B.7 Function Inlining

```
#include "llvm/Pass.h"
#include "llvm/Support/CallSite.h"
#include "llvm/Function.h"
#include "llvm/Instructions.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/Transforms/Utils/Cloning.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/CFG.h"
#include "llvm/ADT/SCCIterator.h"

#include "MyInlinerPass.h"
#define DEBUG_TYPE "MyInlinerPass"

#ifndef INLINE_LOOP_SIZE
#define INLINE_LOOP_SIZE 200
#endif

#ifndef INLINE_DEF
#define INLINE_DEF 15
#endif

#ifndef INLINE_LEAF
#define INLINE_LEAF 20
#endif

#ifndef INLINE_IN_LOOP
#define INLINE_IN_LOOP 200
#endif

#ifndef INLINE_LEAF_IN_LOOP
#define INLINE_LEAF_IN_LOOP 300
#endif

using namespace llvm;

MyInlinerPass *createMyInlinerPass() {
    return new MyInlinerPass();
}

char MyInlinerPass::ID = 0;
static RegisterPass<MyInlinerPass>
X("MyInlinerPass", "Function inlining");

STATISTIC(NumInlined, "Inlined functions");
STATISTIC(NumDef, "Inlined default");
STATISTIC(NumInLoop, "Inlined in loop");
```

```
STATISTIC(NumLeaf, "Inlined leaves");
STATISTIC(NumLeafInLoop, "Inlined leaves in loops");

MyInlinerPass::MyInlinerPass() : ModulePass(&ID), Func(0) {}

void MyInlinerPass::getAnalysisUsage(AnalysisUsage& AU) const {
}

void MyInlinerPass::setBuiltFunction(Function *F) {
    Func = F;
}

bool MyInlinerPass::runOnModule(Module &M) {
    assert(Func && "Running MyInlinerPass without registered Function");
    bool Changed = false;
    Modified.clear();

    Changed |= inlineCalls();
    Changed |= inlineCallers();

    return Changed;
}

std::set<llvm::Function*>& MyInlinerPass::getModified() {
    return Modified;
}

bool MyInlinerPass::inlineCalls() {
    bool Changed = false;
    std::vector<CallInst*> Calls;

    for(Function::iterator FI = Func->begin(), FE = Func->end();
            FI != FE; ++FI) {
        for (BasicBlock::iterator BI = FI->begin(), BE = FI->end();
                BI != BE; ++BI) {

            if (CallInst *C = dyn_cast<CallInst>(BI)) {
                Calls.push_back(C);
            }
        }
    }

    for (unsigned i = 0; i < Calls.size(); ++i) {
        Changed |= visitSite(Calls[i]);
    }

    return Changed;
}

bool MyInlinerPass::inlineCallers() {
    bool Changed = false;
    std::vector<CallInst*> Callers;

    for (Function::use_iterator UI = Func->use_begin(), UE = Func->use_end();
            UI != UE; ++UI) {
        if (CallInst *C = dyn_cast<CallInst>(UI)) {
            Callers.push_back(C);
        }
    }

    for (unsigned i = 0; i < Callers.size(); ++i) {
        Changed |= visitSite(Callers[i]);
    }

    return false;
}

static bool isLeaf(Function *F) {
    for (Function::iterator FI = F->begin(), FE = F->end(); FI != FE; ++FI) {
        for (BasicBlock::iterator BI = FI->begin(), BE = FI->end();
                BI != BE; ++BI) {
            if (CallInst *CI = dyn_cast<CallInst>(BI)) {
                CallSite CS = CallSite::get(CI);
                if (CS.getCalledFunction()->getName().startswith("Func"))
```

```
                    return false;
                }
            }
        }

        return true;
}

static unsigned funcSize(Function *F) {
        unsigned size = 0;
        for (Function::iterator FI = F->begin(), FE = F->end(); FI != FE; ++FI)
            size += FI->size();

        return size;
}

static bool isInLoop(BasicBlock *BB, Function *F) {
        for (scc_iterator<Function*> I=scc_begin(F),E=scc_end(F); I!=E; ++I) {
            if (I.hasLoop()) {
                unsigned size = 0;
                bool inLoop = false;
                const std::vector<BasicBlock*>& Nodes = *I;
                for (unsigned i = 0, e = Nodes.size(); i != e; ++i) {
                    if (BB == Nodes[i]) {
                        inLoop = true;
                    }
                    size += Nodes[i]->size();
                }

                if (inLoop && size <= INLINE_LOOP_SIZE) {
                    return true;
                }
            }
        }

        return false;
}

bool MyInlinerPass::visitSite(CallInst* C)
{
        CallSite CS = CallSite::get(C);
        Function *Callee = CS.getCalledFunction();
        Function *Caller = CS.getCaller();

        if (!Callee->getName().startswith("Func"))
            return false;
        if (Callee->empty())
            return false;

        bool Inlined = false;

        unsigned Size = funcSize(Callee);
        bool Loop = isInLoop(C->getParent(), Caller);
        bool Leaf = isLeaf(Callee);

        bool Inline = false;
        if (Size <= INLINE_DEF) {
            Inline = true;
            ++NumDef;
        } else if (Leaf && Size <= INLINE_LEAF) {
            Inline = true;
            ++NumLeaf;
        } else if (Loop && Size <= INLINE_IN_LOOP) {
            Inline = true;
            ++NumInLoop;
        } else if (Loop && Leaf && Size <= INLINE_LEAF_IN_LOOP) {
            Inline = true;
            ++NumLeafInLoop;
        }

        if (Inline) {
            Inlined = InlineFunction(C);
            if (Inlined) {
                ++NumInlined;
```

```
            Modified.insert(Caller);
            return true;
        }
    }

    return Inlined;
}
```

Bibliography

[1] Clang: A C language frontend for LLVM, nov 2009. `http://clang.llvm.org/`.

[2] LLVM language reference manual, nov 2009. `http://www.llvm.org/docs/LangRef.html`.

[3] Unladen swallow, nov 2009. `http://code.google.com/p/unladen-swallow/`.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006.

[5] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[6] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences—a method to expedite the evaluation of closed-form functions. In *ISSAC '94: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249, New York, NY, USA, 1994. ACM.

[7] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.