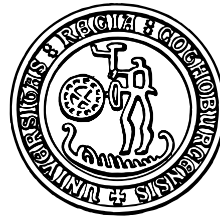
Thesis for the Degree of Master of Science (20p)

# Tuning Intel x86 Executables

**Håkan T. Johansson**

**Supervisor: Håkan Sundell**

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, December 2002

A dissertation for the Master Degree in Engineering Physics at
Göteborg University and Chalmers University of Technology

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2002

**Abstract**

This thesis describes an approach to do post-compile tuning on computer programs in an attempt to increase their execution speed, directed at 32-bit executables for the Intel x86 platform running GNU/Linux or Windows. The plan is to first read the executable file, disassemble the instructions and analyse them to recover the information that was lost after compilation but is needed to later modify and recreate the program. Then various transformations of the program's instructions are attempted. Finally a new executable file is written.

This work consists of two parts. Firstly to identify the problems involved, particularly with the code analyser, which is the complicated part of the system. Secondly to implement a system performing the described procedure.

The results are that no problems preventing a successful analysis has been found, provided the original program is reasonably well behaved. The implementation has been completed so far that a few very simple programs can be modified. The reason for not having completed the actual system is the limited time available for a master's thesis.

**Sammanfattning**

Denna rapport beskriver ett sätt att utföra modifikationer på ett färdigkompilerat datorprogram, med målet att öka programmets exekveringshastighet, riktat mot 32-bitars program för Intels x86-plattform körande GNU/Linux eller Windows. Planen är att först läsa in programfilen, disassemblera instruktionerna och analysera dem. Detta för att beräkna den information som gick förlorad efter kompileringen men som behövs för att sedan kunna ändra i och återskapa programmet. Därefter utförs olika transformationer av koden. Slutligen skapas en ny exekverbar fil.

Detta arbete består av två delar. Dels att identifiera vilka problem som är förknippade med den beskrivna proceduren, särskilt de som rör kod-analysatorn, vilken är den komplicerade delen i systemet. Dels att implementera ett system som utför den beskrivna proceduren.

Resultaten är att inga problem som omöjliggör analys har hittats, under förutsättning att originalprogrammet uppför sig tillräckligt väl. Implementationen är så pass färdig att den kan hantera ett fåtal väldigt enkla program. Anledningen till att implementationen inte är färdig är den begränsade tiden för ett examensarbete.

# Contents


**1 Introduction** **1**
- 1.1 Background and motivation . . . . . . . . . . . . . . . . . . . . . 1
  - 1.1.1 Acceptable input programs . . . . . . . . . . . . . . . . . 2
- 1.2 The x86 processor family . . . . . . . . . . . . . . . . . . . . . . 2
  - 1.2.1 Data storage . . . . . . . . . . . . . . . . . . . . . . . . 2
  - 1.2.2 Instructions - assembler . . . . . . . . . . . . . . . . . . 4
  - 1.2.3 The CPU generations . . . . . . . . . . . . . . . . . . . . 5

**2 Strategy** **6**
- 2.1 Reading the executable . . . . . . . . . . . . . . . . . . . . . . . 6
  - 2.1.1 Contents of an executable file . . . . . . . . . . . . . . . 7
- 2.2 Disassembling the instructions . . . . . . . . . . . . . . . . . . . 8
  - 2.2.1 Decoding one instruction . . . . . . . . . . . . . . . . . . 10
  - 2.2.2 Disassembly loop . . . . . . . . . . . . . . . . . . . . . . 10
- 2.3 Data flow analysis . . . . . . . . . . . . . . . . . . . . . . . . . 11
  - 2.3.1 Analysis loop . . . . . . . . . . . . . . . . . . . . . . . . 13
  - 2.3.2 Analysing one instruction . . . . . . . . . . . . . . . . . . 13
  - 2.3.3 Circular dependencies . . . . . . . . . . . . . . . . . . . . 15
  - 2.3.4 Pointers . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  - 2.3.5 Data structures . . . . . . . . . . . . . . . . . . . . . . . 16
  - 2.3.6 The alias problem . . . . . . . . . . . . . . . . . . . . . . 17
- 2.4 Writing a new executable . . . . . . . . . . . . . . . . . . . . . . 18
  - 2.4.1 Reassembly of instructions . . . . . . . . . . . . . . . . . 18

**3 Tuning** **19**
- 3.1 Inlining . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
- 3.2 Streamlining . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
- 3.3 Avoiding jumps . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
- 3.4 Reordering . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
- 3.5 Rearranging data structures . . . . . . . . . . . . . . . . . . . . . 21
- 3.6 Going the other way . . . . . . . . . . . . . . . . . . . . . . . . . 22

**4 Implementation** **23**
- 4.1 Storage classes . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
- 4.2 Working classes . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

**5 Results** 27
5.1 Implementation . . . 27
5.2 Reading and writing executables . . . 27
5.3 Changing an executable . . . 27

**6 Finale** 28
6.1 Conclusions . . . 28
6.2 Future work . . . 28
6.2.1 To do . . . 28
6.2.2 User feedback . . . 29
6.2.3 Other uses . . . 29
6.3 The source . . . 30

**Acknowledgements** 31

**A Motivator** 32

**B Short introduction to x86 assembly** 33

**C Execution stack at work** 37

**D PE and ELF files** 39
D.1 PE file (executable for Windows) . . . 39
D.2 ELF file (executable for GNU/Linux) . . . 40
D.3 Similarities . . . 41

**E Data flow analysis example** 42
E.1 Partial registers . . . 42

**F Tuned examples** 44

**Bibliography** 46

**Glossary** 47

# Chapter 1

# Introduction

The aim of this project is to construct a tuner - a program capable of reading an executable program, decipher its low level construction, make adjustments to it and write an equivalent executable program. The goal is that the output program should run faster than the original. With equivalent is meant that the generated program should behave exactly as the original, except for execution speed and perhaps used memory.

A glossary of words written in *italics* is at the very end of the thesis.

## 1.1 Background and motivation

Normally, computer programs in machine readable form are created by a number of source files being translated (compiled) by a *compiler* into object files which are then merged into an executable file by a *linker*.

Compilers can make many optimisations[1] of the generated code, based on all the information in each source file. But linkers usually collect the code and create a program without any attempts to improve the final code, e.g. by simplifying function calls or removing unnecessary *register-stack operations* to retain variables during calls to subroutines, not to mention doing any *inlining*. Linkers cannot do these things simply because the information necessary was lost after the source was translated into object files. Appendix A demonstrate this problem.

The appropriate and best solution to this problem (the blind linkers) would be to perform whole-program optimising compilation, where the code generation is moved from the compile to the link stage, so advantage can be taken of the information about the entire program when creating it. This work is an attempt to tackle the problem from another direction, a somewhat backwards direction, since the plan is to adjust the final, compiled program without access to the source and the information therein that was lost. Most of the information lost must therefore be recovered by analysis of the program. The most important loss, i.e. hardest to rectify, is the lost information about individual data structure sizes, see Section 2.3.5.

[1] Not really optimisations, but improvements, because optimum is to be *the* best. However, optimisation is the word in general use for this.

### 1.1.1 Acceptable input programs

The tuner is meant to be able to handle most well-behaved, compiled C programs. A program is well-behaved when it does not do any strange modifications to the execution path. The C functions `setjmp` and `longjmp` does this and self-modifying code is another untreatable construction. This means that only normal jumps and function calls are allowed, including function pointers and indexed jumps. A program with inline *assembler* may be well-behaved, provided the execution path is not modified strangely. Actually, the tuner does not assume that the input program was written in any specific language, so C is not a requirement, just an example.

This project will deal with programs compiled for the x86 family of processors, simply because I have such hardware available, and would therefore also benefit from a completed system. As the problems involved stem from the executable code itself, and not the packaging, both 32-bit Windows and GNU/Linux executables will be treated.

## 1.2 The x86 processor family

This project is aimed at the Intel x86 processor family, with the first members produced over 20 years ago. Compatible processors are also produced by AMD, VIA and Transmeta.

Before describing the idea of the thesis, an introduction to the most important concepts is in order. A computer program operate on data according to its instructions. The instructions are stored in memory and executed sequentially, one after another. At the low (machine code) level this work is at, data is only integer and floating point values. High level constructs like structures (which are compounds of the low level data types), do not have special instructions but are handled by more or less complicated combinations of the instructions available for manipulating the simple data types. So there is no need to deal with the complexities of different programming languages, as everything is encoded with the x86 instruction set anyway, and that is the only thing that has to be understood.

### 1.2.1 Data storage

Variables (data) are stored in memory during execution of a program. There are three major places to store variables depending on their lifespan and usage:

#### Main memory

The memory is located outside the processor and organised as a long list of bytes. When the program want to read or write something it references the location with an integer as the address, an offset into the list. Data in main memory can be kept during the entire execution time of a program.

#### Registers

Variables that are accessed frequently (e.g. partial results during the evaluation of an expression, like $a + 5 \cdot (b + c)$, which must be calculated using several

instructions) are stored in registers, which is memory inside the processor. Registers can be referenced easily, because there are special encodings for each register for most instructions, while for memory there is only a few encodings for accessing memory, and then extra information is needed to know what address is desired. Registers are also accessed very quickly because they are inside the processor, but they are few. So while executing, the at the moment most important stuff is kept in registers, and when a particular piece of data is no longer needed, any result that is to be remembered is stored in memory (perhaps on the stack), so that other data can be kept in that register.

Compared to other architectures (like Sparc and Alpha processors), the x86 family has relatively few registers (only 8 general purpose: `eax ebx ecx edx esi edi ebp esp`), but can on the other hand operate directly on memory (add and subtract etc.), so not everything has to go into a register before being used. See Appendix B or [1, 2] for more details.

### Execution stack

One part of the main memory is used as a *stack* during execution. The stack follows execution in the respect that as the program gets deeper into function calls, the used stack space grows, and when the functions return, the stack shrinks.

The stack is a scratch-pad during execution of a function for storing local variables, because they are normally more than the available number of registers (which can happen even for a processor with many registers). The execution stack is normal memory, however especially allocated by the program for use as scratch space. The amount of stack space used is tightly connected to what function and instruction in the program that is currently running because its use is hard coded in the program, as opposed to other memory which is dynamically allocated to hold user data and whose size may vary between runs. One of the general purpose registers always point to the top of the stack, the stack pointer, `esp`[2].

Data stored on the stack can be accessed as any other memory via pointers with the address of the data. This address is normally relative to the top of the stack, `esp`. The `push` instruction increase the stack space and store one value at the top of the stack. The opposite is done by `pop`, which copy the value at the top of the stack, and then decrease the stack space. Both these instructions use and modify `esp`, without the need to specify it as the memory address.

The stack is also used to store the return address when a function is called. That is, the `call` instruction (used by the caller) pushes the address of the instruction following the call (where execution should continue when the called function is finished) before jumping to the called function. At the end of the called function, a `ret` instruction is executed, which reads and pop the topmost entry on the stack and uses it as a jump target. This of course require the function to increase and decrease (push and pop) the stack in equal amounts, so that it is the return address that is on top when `ret` is issued. Programs not doing this, or in other ways modifying the return address are not well-behaved, as specified in Section 1.1.1.

[2] It can be used for other calculations as well, but an interrupt handler could then easily make the program crash, if it assumes that `esp` hold a valid stack pointer.

As the execution stack is important for this work, Appendix C has a clarifying example.

### 1.2.2 Instructions - assembler

After introducing some stack-modifying instructions in the previous Section, it is also in order to present some other common x86 instruction types and examples of them[3]:

- Copy: `mov`
  Copy a variable from one storage location to another, e.g. from memory to a register or the opposite.

- Arithmetics: `add, sub, cmp, inc`
  Perform elementary mathematical operations, like add, subtract, multiply and divide. `cmp` does a subtraction without storing the result, only the status flags are affected. This is used for comparisons. `inc` is a short form to add 1 to a register or memory location.

- Jump: `jmp`
  Unconditionally jump and continue executing instructions at another location.

- Flow control: `jl, jnl`
  Do a conditional jump, based on some selected status flags in the processor. The status flags in turn were set based on the outcome of the most recent arithmetic calculation. So one can for example calculate the difference between two values by subtraction and make a jump depending on if the result (the difference) was less than zero, which is the same as the first value being less than the second. So the `cmp, jl` pair implements a $<$ comparison.

- Procedure: `call` and `ret`
  The call instruction unconditionally jump to the specified target, but also make a note in the stack of were the next instruction after the call is. When a return instruction is executed, the top of stack is popped and execution continued at that address, which (if not messed with) is the instruction directly following the `call`. This way the same code can be used from several locations. It is a function. It may for example be capable of counting the words in a sentence, make the speaker beep, or open a file.

A short description of the instructions used in this thesis can be found in Appendix B.

[3]This even if this report try to avoid both the gory details of direct processors programming (assembler) and the inner workings of the implemented tuner itself, for which the reader is referred to any available documentation (e.g. [3, 4, 5]) and the source (see Section 6.3), respectively.

### 1.2.3 The CPU generations

For our purposes the description may start from the middle of the history with the 386 chips, the first 32-bit capable ones. This chip usually need several clock cycles to perform any one instruction.

Then came the 486 chip with a better *pipeline*, which is capable of a throughput of one instruction per clock for the simplest (and most used) ones.

After this the 586 (Pentium) arrived, with a dual pipeline, in principle a dual 486, with the capability of executing two simple instructions simultaneously, provided they do not depend on each other.

The x86 assembler is called a CISC architecture, because its instructions are capable of operating directly on memory operands (and not only registers) and there are a lot of special instructions. The more complex instructions can usually be replaced by a sequence of several simple instruction. Another approach to processor design is deployed by RISC architectures, who use a more limited instruction set. Their instructions cannot operate directly on memory operands, so all needed data has to be loaded to a register before being used. They usually have more registers making it easier to avoid close dependencies of instructions, and the smaller instruction set simplifies chip design.

The CISC plethora of different instructions and operations require a lot of hardware for decoding and execution. With the 686 (Pentium Pro, II), translation of the instructions into a smaller set of simpler ones ($\mu$ops, basically a RISC set of instructions) inside the processor is used to simplify the calculation core of the processor. It is also capable of out-of-order execution of these simpler instructions.

Out-of-order execution sounds magical, but it really only is to put the $\mu$ops into a queue of instructions to be executed, and because they are simple, it is easier to determine any dependencies. Then the execution units fetch instructions from this queue beginning at the top. But sometimes (when the top instructions are waiting for dependencies), instructions further down the list that do not depend on any other not yet executed instruction can be executed. Then there is of course some extra circuitry to make sure that the result is the same as if the instructions were executed in order. This is important should something happen which has to stop execution after a specific instruction and then do something unanticipated (e.g. taking an unforeseen branch). Then all instructions up to that one have to be performed, and no $\mu$ops enqueued after it may affect the state of the processor.

# Chapter 2

# Strategy

The first idea was to do some small changes to the executable code of a program (i.e. *peep-hole transformations*). However, it is quickly realised that this cannot be done to an executable, because even very simply transformations that increase code size or move instructions the slightest may affect pointers to that code, making changes throughout the entire file necessary. So even if in some cases some small changes can be made without breaking an executable, in order to make any significant changes a full-scale understanding of the possible execution paths is needed, so that they can be preserved.

So to be able to do any modifications at all, a basic administrative system is required: reading, disassembling, analysing and writing executables. The different stages are shown in Figure 2.1. The difficult part is the analysis, the other operations are straightforward as they deal with strictly specified file and instruction formats.

To have a chance of successfully altering a program without destroying it, two things are required:

- A list of the instructions in the program along with information of how they can follow each other due to jumps, calls and returns. This is explicitly present in the executable file. It just have to be transformed into a more useful (changeable) representation.
- A good description of the low level data structures used. This information is implicitly specified by the instructions, and must be extracted with much more pain. Without this information only very limited changes can be made to the program, because no memory accesses can be modified.

This knowledge is gained by disassembling and analysing the program.

## 2.1 Reading the executable

An executable is a file which is read and setup into memory by the operating system loader when executed by the user. With some minor adjustments the image is copied as is into memory. The adjustments are mainly the binding of

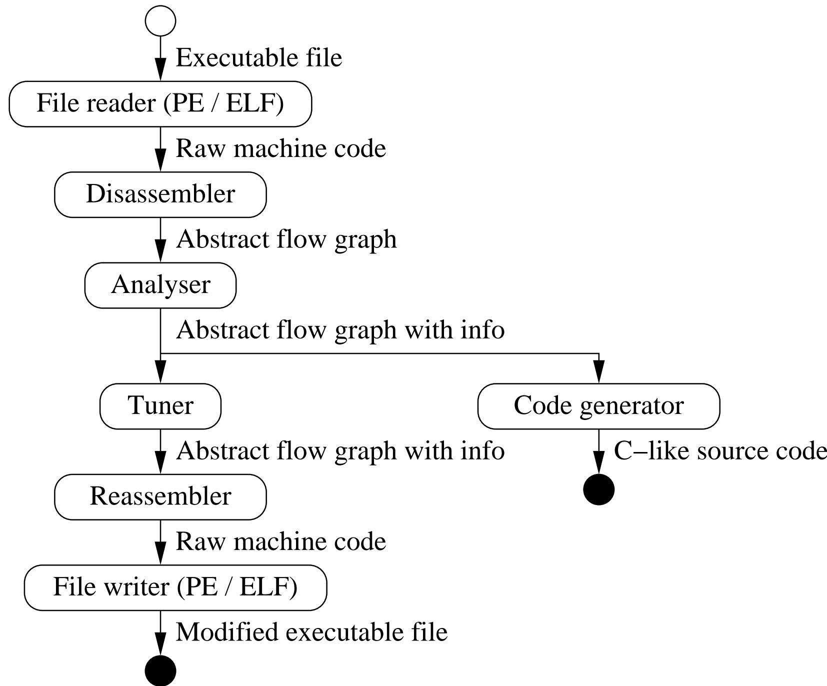



Figure 2.1: Flow of an executable through the tuning system. The auxiliary output on the right (which is not part of this work) is to show that the output from the analyser could be fed into a decompiler output stage, see Section 6.2.3.

import libraries to the executable image. Import libraries may provide runtime functions such as `printf`[1], gui functions or routines for manipulating images.

### 2.1.1 Contents of an executable file

As previously mentioned, an executable file is basically a copy of the program image in memory on start, along with information on how to setup this image enclosed in some packaging.

The parts of the executable file of most interest to this project is the actual code and any initialised data. Of importance is also information on imported and exported functions and variables. All this information is treated by the disassembler/analyser to extract the execution behaviour of the file.

Any other stuff in the executable is simply read and stored so it can be written to the new executable. As this program do not know how to handle such information, i.e. how it is formatted, this will work only if that information is not destroyed by being *relocated*. One trick would be to never move unknown sections, possibly by splitting any sections that would cause this (i.e. sections before them that have grown) into one part using the old space, and another part located after the possibly immobile section.

[1]The beauty of C.

For a more elaborate description of the executable file formats used, see Appendix D and [6, 7].

## 2.2 Disassembling the instructions

Before changes can be made to the input program, the instructions must be lifted from the rigid structure in the executable image to a more flexible representation, which of course will use much more memory, but be editable. As shown in Figure 2.2, the representation is a graph, with nodes representing

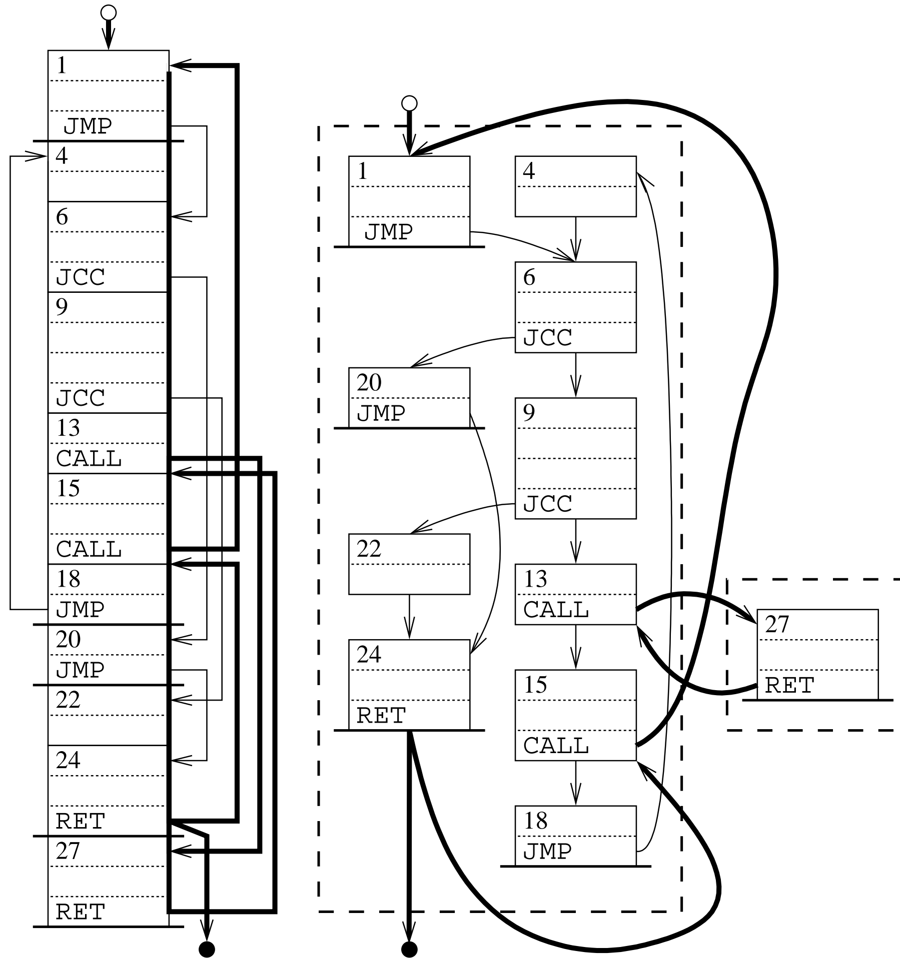



Figure 2.2: Instructions in the rigid structure of an executable to the left, and as the internal graph representation to the right. Each solid rectangle is a trace. Traces end with an instruction affecting execution flow, or because the next instruction is a jump/call target (since a trace can only be entered at the top). Two functions (enclosed in dashed rectangles) have been identified. The larger one call the smaller one and itself recursively.

instructions and edges representing possible execution paths. The operands of the instruction objects, are also objects, so they easily can be created, inserted and deleted into the structure with small or no effects on the surrounding. Specifically, the intermediate format is not dependent on the address the instructions occupy like the raw machine-readable format in the executable image (e.g. relative jumps).

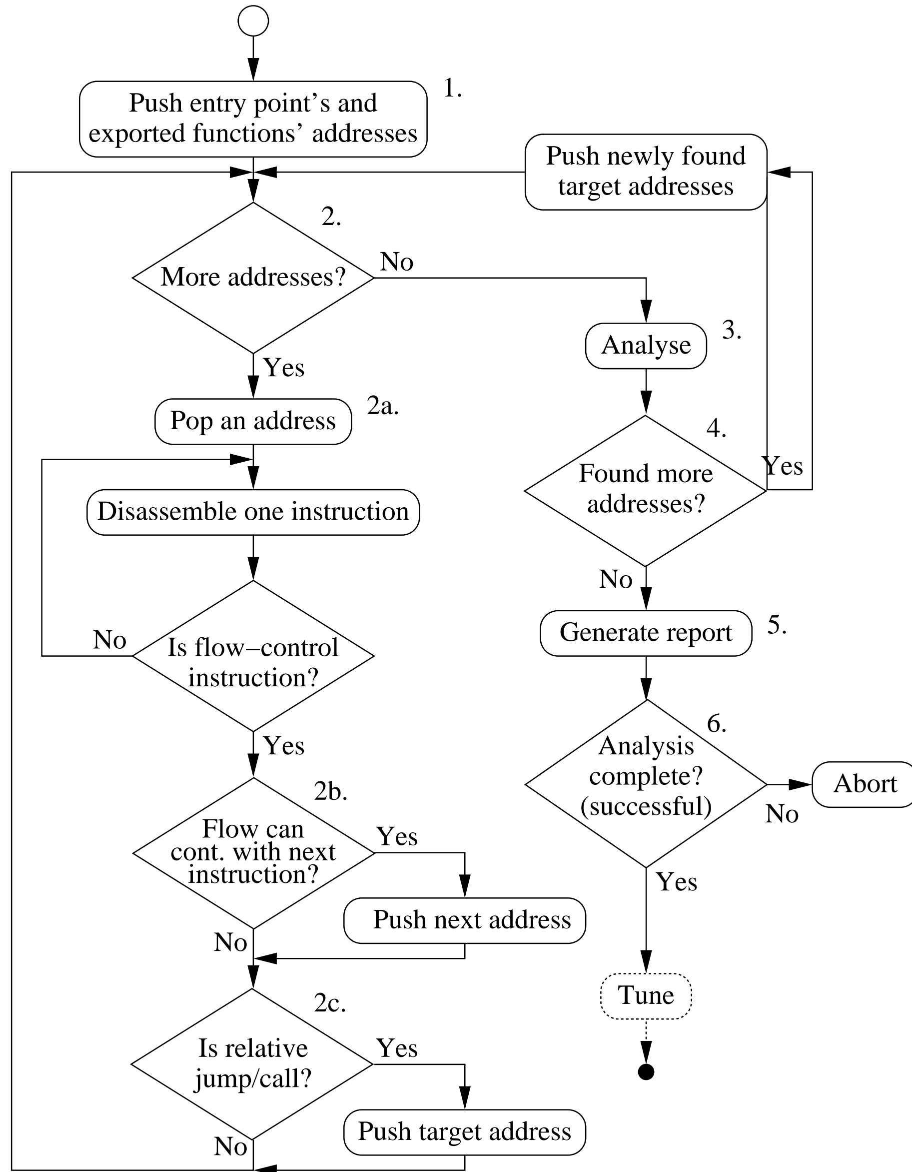



Figure 2.3: Flow chart of the disassembly loop (see next page). If the analysis is successful, i.e. everything needed to be known has been found out, it is possible to continue with tuning the disassembled code.

### 2.2.1 Decoding one instruction

The disassembly of the x86 machine instructions is performed with a table driven decoder, i.e. each instruction byte is used as an index into a table, which then tell what instruction it is and how it is encoded. This design make the disassembler easy to adjust (e.g. to add new instructions). Due to the complexity of the x86 instruction set, with variable sized instructions, and many special instructions, there are several layers of decoding tables and some special cases. The approach with the bulk of information in tables (i.e. data instead of code) also simplifies the reassembler considerably, see Section 2.4.1.

### 2.2.2 Disassembly loop

Recipe for executable code collection, also see Figure 2.3:

1. The program's main entry point is found by reading the executable's headers, and any externally visible functions are found by reading the export tables, if present. The entry-points' addresses are pushed onto a *stack* of addresses to be disassembled.

2. While the decoding address stack has elements:

   (a) Popping one address of the stack at a time, small portions of code are disassembled (traces). Disassembly of a trace is ended when a flow control instruction is found: (un)conditional jump, call or return. This way, disassembly follows the normal execution path.

   (b) If the execution may continue after the last instruction (call or conditional jump), the continuation address is pushed onto the stack.

   (c) If it is a relative jump or call, the target address is easily calculated and pushed onto the stack. Otherwise, calculation of the target is deferred until later analysis.

   However, if the jump or call target is inside an already disassembled trace, the old trace is split at the target. If the target inside the trace is not at the first byte of an instruction (which could happen as we have variable length instructions), we call it a crazy jump, an error[2], see Figure 2.4.

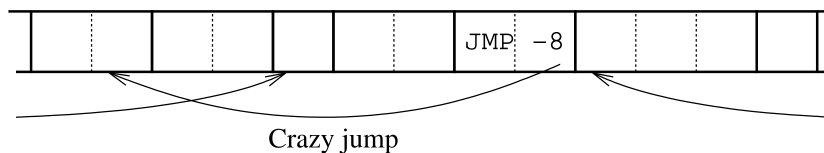



Figure 2.4: Two sane jumps and one crazy jump targeting the middle of an instruction. The target byte then has multiple instruction interpretations.

3. Now the data flow of the disassembled code is analysed, see Section 2.3. We may find some information affecting the execution of the code:

[2]It is not an error for a program to be constructed this way, but it is highly unusual and not supported by this system, as it would give instruction bytes dual interpretations.

- Target addresses encoded using absolute addressing (which always uses a pointer to memory or a register holding the actual address). This may come from switch statements, or be function pointers.
- Unreachable code that was disassembled anyway. An example is the code following two conditional jumps, jumping on opposite conditions (e.g. `je` and `jne`), and without the second jump being a branch target, see Figure 2.5. The disassembled but unreachable code is discarded.

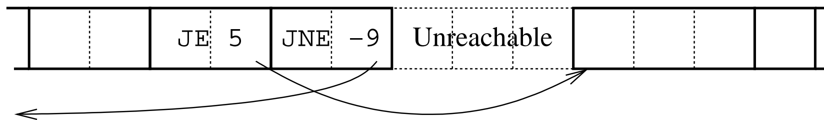



Figure 2.5: Unreachable code after two conditional jumps. The code cannot be reached as the conditions exhaust all possibilities, and there is no path that only try the second condition.

4. If any new targets in the executable were found, they are pushed onto the stack and disassembly is resumed, starting at step 2 again.
5. Optionally, a detailed report of the executable is generated to give a view of what is going on. It is generated as several HTML pages and include the executable file's headers, code disassembly and data structures used in the program.
6. The disassembled code is checked so that there are no "crazy jumps" and that all dynamic jumps and calls have all their possible targets disassembled. If not, treatment of the program is aborted with an error message after generating the report, because no tuning can be performed on a program only partially understood.

| | |
|---|---|
| *Face:* | *Well, that concludes this part of the fact finding mission.* |
| *Hannibal:* | *But don't be sad, we saved the best part for last.* |

## 2.3 Data flow analysis

| | |
|---|---|
| *Hannibal:* | *Phase one of the operation is a success.* |
| *Face:* | *Well, phase one is always the easiest.* |
| | *The bad guys never know it when you're doing phase one.* |
| *Hannibal:* | *Haha* |
| *Face:* | *Phase two, that's when the soup sticks to the spoon.* |
| | *the A-team, episode: Pure-Dee Poison* |

Phase two of interpreting the executable is data flow analysis. The primary goal of this is to find and trace all information affecting the execution path and memory reads and writes, i.e. tracking all pointers and their use and knowing how, when and where the stack grows and shrinks.

A secondary objective is to find out as much other static information about the variables (global and local (stack and registers)) as possible. With static

information is meant knowledge about the possible values of variables during execution. For example a boolean (true/false) variable should be known to only be able to get the values 0 and 1 during execution, even when it is stored in a place that can represent a larger range. Static information can be used to make more clever transformations to the code in the later tuning steps, perhaps even remove unused parts of the code. Much of the static information also come for "free", because it is anyway needed to be able to reach the primary goal of tracking the pointers.

The term variable is used in a broader sense than normal, because also locations holding a partial result for calculating a larger expression is considered a variable, and treated in the same way as any other data in memory, on the stack or in a register. Three kinds of information is collected about variables in the program:

- Actual content, i.e. possible values. This keep track of constants, sets of constants, ranges (with a minimum and maximum value), patterns (where some bits are known to be either 0 or 1, the other unknown) and of course unknowns (when it has not been possible to infer any information on the contents of the variable from the operations that led to it). If, for example, an unknown value is taken `and`[3] a constant, any bits being 0 in the constant will also be zero in the result, while the rest still are unknown, yielding a pattern. When possible, for any instruction operating on some known (or unknown) value, it is attempted to create a known value as output so as to reduce the uncertainty of the variable's contents. Knowing the possible values of a variable, is the same as also knowing what values a variable not can have, which for some tunings may be useful.

- Abstract, semantic content. Even if the actual value of a variable is not known, it may represent something important affecting execution. This includes pointers to data or functions and a pointer to the execution stack. Usually, the actual value of a stack pointer is unknown, but it is known as an offset into the stack, relative to the location of the stack top at function entry.

- Origin. Every variable in a function has an origin, either being a function argument, coming from a memory read, being set from a constant or as a result of a calculation. By creating a record of how data is flowing through the program it is possible to find the sources of variables. This is important for pointers, because sometimes it is first when a variable is used as a pointer it becomes apparent that it is a pointer. One can then trace it backwards through the program to locate the instructions setting its value(s) (e.g. constants), so they can be safely and properly relocated.

  When a variable is the result of a calculation, it is recorded as depending on the calculation (e.g. addition) which in turn depend on the source variables. This way, one can also reconstruct larger mathematical expressions that have been broken down into their constituents.

As yet, no changes can be made to the code based on our new knowledge (e.g. constant propagation or changing unnecessary conditional jumps

[3]Bitwise and: each bit in the result becomes 1 iff both source bits are 1, otherwise 0.

to unconditional), because the analysis may be followed by more disassembly passes (if more code was found via jump/call targets),

One preliminary change is however necessary: unreachable code removal. This is done by inserting a dead code marker so the disassembler won't try to decode it again. Code can be unreachable because of conditional jumps as explained in Figure 2.5, and because of function or system calls that never return. For example, on Linux, the software interrupt `int 80` with system call number 1 in register `eax` will terminate the current process and any following instruction will never be executed.

Removal is necessary both because bytes following the last reachable instruction never were intended as machine code and therefore most probably represent random instructions. Analysing this junk would pollute the analysis of the rest of the program with unnecessary (and probably inconsistent) restrictions. Not removing wrongly disassembled code could also make the disassembler unable to disassemble correct code, as it may then wrongfully consider a target inside the incorrect code a crazy jump, see Figure 2.4.

### 2.3.1 Analysis loop

The code is split into small fragments (traces) by the disassembler, see Figure 2.2. A trace can only be entered at the beginning and branching can only occur at the end. For each trace it is known from where execution may come, and where it may proceed.

The traces are collected into blocks called functions (because the collections usually correspond to high-level functions). Functions can only be reached by `call` instructions, and left by `ret` instructions. Inside each function there are usually jumps between the traces. It is possible for a function to have multiple entry points if some (common) trace in the function can be reached from several call-targets (via jumps). This may cause trouble when analysing, if the common trace is reached with different depths of the local stack depending on the entry point.

Analysis is performed repeatedly until the result is self-consistent, see Figure 2.6. All functions are put on a list of functions to be analysed, and then handled one at a time. A function may be put on the list again after having been analysed, if analysis of a caller or callee changed the known information about the state when the function is called or makes a call.

When a function is analysed all traces within it are analysed. For each trace, all the inputs (all values of the registers and stack) are set to the least common knowledge of all the traces' outputs that may pass control to this trace. Then the trace is investigated and all statically known information in the trace is calculated and set as its output. Next all known targets are examined to see if the new information gained will affect any target trace input so that the input knowledge has changed. If so, the target trace is put up for reanalysis.

### 2.3.2 Analysing one instruction

Each instruction is inspected by means of an abstract execution of the instruction, a simulation taking into account all possible input values at the same time. A record is kept of the known changes to the processor state after each instruction. Before an instruction is inspected, the analyser get any source variables

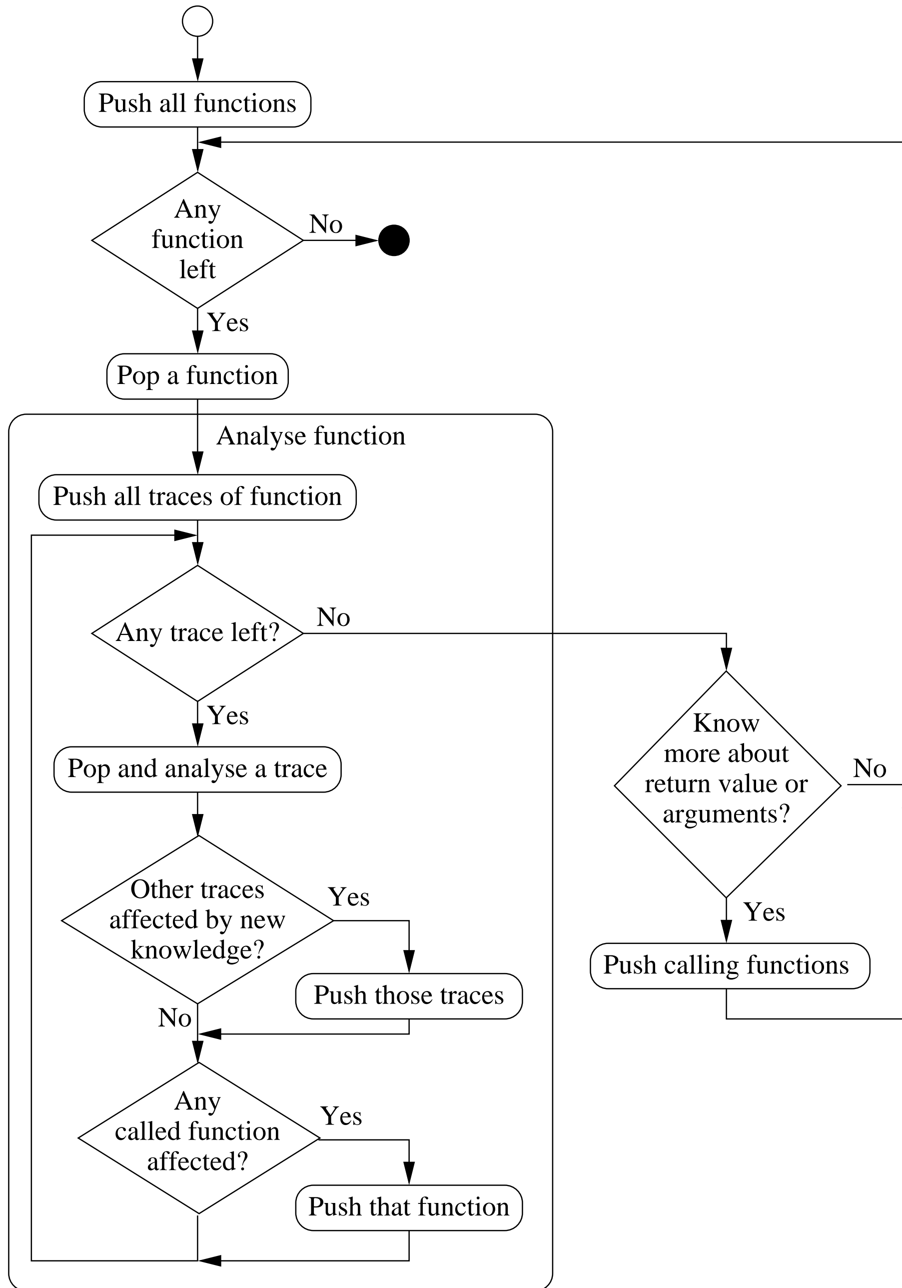



Figure 2.6: Flow chart of the analysis loop.

from the state record. It is searched backwards to the most recent change of the register or stack location requested. Access to memory other than the stack is handled in a different manner, since accesses to memory are of an unordered nature - e.g. a read strictly before a write inside a function could occur after the

write, if the function is called several times.

If the search for a source operand is unsuccessful all the way back to the entry of a function, the requested operand will be noted as an argument to the function. After a function has been analysed, a check is also made to see if any outgoing register state has been restored to the initial one. This is the case if a register has been used as a local variable, and the original value was saved (e.g. via a pair of `push` after entry and `pop` before return) during the function call. Such a register is not an argument, unless it was also otherwise used in any calculation.

An example of a simple analysis of a function is in Appendix E.

### 2.3.3 Circular dependencies

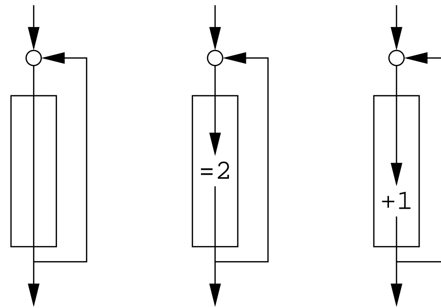



Figure 2.7: Three variables passing through a loop. The rectangle is the loop body, and the circle represent merging of the possible incoming values. The leftmost variable is unaffected by the loop, the middle one assigned and the right has a circular dependency on itself.

Anytime a trace inside a function can be reached by itself, such as in a loop, any variable set by an instruction in that trace may have a dependence on itself, see Figure 2.7. If it is just set in the loop, there is no problem, and the possible values are just merged with the possible values from other parts of the function reaching the loop. However, a loop counter for example would depend on itself and could create an infinite dependency chain if not treated specially, and the chain broken.

Unfortunately, the obvious and easiest solution, to assume an unknown incoming state at loop entry until all source branches' outgoing states (including the one from the loop) are known would be disastrous. Firstly, because nothing would ever be known as the loop itself cannot be analysed and give an useful outgoing state, as the incoming state is unknown. Secondly, as the stack state would become unknown. Thirdly, because variables not being touched by the loop, but used again after it would be lost, for no good reason.

Instead, all loops have to be analysed several times. At the entry, only the states from source branches so far analysed are used. When the loop has been analysed, the state from its outgoing branch is also known, and the loop can be reanalysed using that result too. To prevent variables depending on themselves from causing infinite reanalysis, such dependencies are detected, and those

variables are recorded as having an unknown state on loop entry. And since an unknown variable state will merge with any other state (constant, range or pattern) to an unknown state, infinite analysis is avoided.

A similar problem is created by recursive functions. Also for this case the second solution must be used, because arguments that are not circularly dependent can be pointers, and must be properly tracked. Otherwise the first solution could have been used (at the cost of less knowledge) since the problem of the lost stack pointer is not an issue here because the semantic content of the stack pointer is assumed and set at the function entry point when analysing.

### 2.3.4 Pointers

The data in the variables (registers, stack or memory) flowing through the program can be representations of many things, many which we will never (need to) know the meaning of. One kind that however is of crucial interest is the one that directly affect program execution: data and function pointers (addresses of data in memory and jump/call targets). All other data handling will in the simplest approach (without doing any tuning) just be transferred to the output program as is, without destroying the integrity of the code. But data that is used as pointers must be traced through the program backwards to every possible origin. This is done so that every location referring to any object (function or variable) that will be moved as a result of the tuning process, can be changed accordingly. It is also done to find jump/call targets that are only reached via pointers (which are usually set by simple `mov` instructions (assignments), but not necessarily close to the `jump/call` instruction).

Actually, this should not pose a too big problem, as everything is traced through execution anyway. Most data will however quite soon be regarded as the completely unknown bit pattern[4], but pointers will usually be (sets of) constants, which are then easily treated. If anything used as a jump/call target when data analysis is finished is an unknown, a pattern or a range, the pointer has apparently suffered some fancy arithmetics and (which although it may be working for the original program) cannot be handled, because the target addresses are thought to span an improbable and possibly huge set of locations.

### 2.3.5 Data structures

Data structures must be reconstructed because they may contain pointers to other structures and pointers to functions, see Figure 2.8. It is the function pointers that are of most interest. When data structures are dynamically allocated, the information about their contents cannot be associated with a specific location in memory (global or stack), but the information will be associated with the respective members of the structures themselves.

Another difficulty is to determine the size of data structures in statically allocated memory[5] or on the stack. The problem of knowing how large objects data pointers reference is important because this information is vital if changes

[4]Otherwise we could constant-propagate everything, and the entire program would be useless.

[5]Allocated on program startup, usually as a part of the executable image.

Figure 2.8: Data structures with pointers. The pointers `A` and `B` show that two pointers must not point to the same place to access the same data. `C` and `D` also illustrate the alias problem.

are to be made to stack usage. It is not difficult to figure out the size for single structures composed of separate objects, but arrays which are indexed are problematic since it will not usually be possible to figure out the limit of the index variable. The extent of a data structure can be guessed by assuming that it end at any location used by another uncorrelated pointer. The loss of knowledge of data structure sizes is the worst loss during compilation. However, the information is not necessarily needed, but lacking it, most tuning cannot be performed as memory and stack use cannot be rearranged.

Dynamically allocated data cause less trouble, as no relocation is needed, and allocating the right size is handled by the analysed program anyway.

#### Import libraries and callback functions

A special problem for disassembly is posed by *callback functions* used by functions imported from libraries. When investigating a program it must be known if any parameter passed to a function in a library actually is a pointer to some code (or data) inside the original program. This can only be achieved by either inspecting the import library itself and see if any argument is used as a target address for a call, or by parsing the include *headers* for the library.

### 2.3.6 The alias problem

The major problem concerning tuning pointers to memory is the *alias problem*. When having an object in memory, it is possible to have several pointers to that same object. The problem is that it is not generally possible to optimise away the use of a pointer to an object (for example a write later followed by a read), if it is possible that something in between (perhaps deep down some function calls) has written to that object. Also, in general, a memory pointer may address an area used for *DMA* to a hardware device, making changes to memory reads and writes a dangerous business. A program running multiple threads or using shared memory could also experience similar problems.

The alias problem does not make tuning impossible, as it can be circumvented by not allowing any tuning that affect the existence or order of any memory reads and writes. However, disallowing such transformations will

make many improvements impossible. Therefore it is beneficial to find situations where the alias problem does not apply. This is for example the case with some uses of the stack, when a pointer to newly grown space has not been given away for unknown use.

## 2.4 Writing a new executable

And to see if we managed not to over-cook the program, an executable file must now be written. The only difficulty is that when writing one part (section) of the file it is not not known where it will end. As a consequence the starting points of other sections are also unknown and therefore it is not possible to write absolute addresses and references within the file when writing sections referencing yet unwritten sections.

But by solving this problem, the problem of *relocation* is also handled. When an import library is loaded, if the loader is lucky[6] the library can be loaded at the address space in memory the image was originally prepared for. But sometimes that space is already occupied and the library has to be relocated before being used. For this to be possible every library (that is relocatable, and they better be) come with a section of relocation information (a list of offsets in the section containing addresses and how to change them to account for relocation).

One solution is to imagine an address in memory where each section will begin when it is written, and write all references to them is assuming those addresses. But for future use all places that will have to be modified when changing the references are noted. When all sections have been generated, all sections' start addresses are simply adjusted (so they are not overlapping in memory when loaded) and all the locations on the list of references are fixed to reflect that. This information can then also be output to a relocation table, enabling the executable loader to relocate the image once more, if necessary. The entire executable is prepared in memory and written to file when all fixups have been applied.

### 2.4.1 Reassembly of instructions

When the changes to the program's instructions have been done, operating on the internal spaghetti representation of Figure 2.2, the instructions must be converted back to machine readable form.

To avoid information duplication (in the disassembler and reassembler), reassembly is done by inverting the tables that are used by the disassembler. When assembling an instruction, the table entries possibly associated with that particular instruction are easily found by means of a reverse lookup table. Then by trying to encode the instruction according to each of table entries the shortest possible machine code is easily found[7]. It is of course also possible to generate opcodes longer than necessary (but with identical meaning) if that would be useful for *aligning* subsequent instructions on some boundary. Alignment may be favourable for instructions that are jump targets since instruction fetch then may be faster.

[6] And sometimes luck can even be arranged for by a clever programmer beforehand.

[7] Instructions have variable length, and many have several possible encodings.

# Chapter 3

# Tuning

As compilers are (perhaps even extremely) good at optimising code inside functions, our primary goal is not to improve that part, but to do other things. And as the compiler is assumed to have made a good job, the general idea is to preserve the original code as much as possible, only making changes when it is quite sure that they carry improvements.

## 3.1 Inlining

As stated in the introduction, one idea is to inline functions that the compiler never got a chance to try because they were defined in another source file than the caller. The obvious cases are functions only called from one location. To avoid making the memory image too large, all other function inlining candidates must be evaluated to see if inlining would be either smaller than setting up the parameters and doing the call or if it is advisable due to a large number of repeated calls from the same location, i.e. inside a loop.

Inlining a function can also open the possibility to do other simplifications to the code. For example, a function that is called with a constant argument may be simplified when inlined. Inlining is done by making a copy of the inlined function at the call location in place of the `call` instruction which is removed and converting any `ret` instruction to a `jmp` targeting the instruction originally following the call instruction (except any `ret` instruction at the very end of the inlined function, which is simply deleted). When the function has been inlined, other tuning operations can be applied to the new, combined function.

If all invocations of a function has been inlined and the function itself is not exported or a pointer to it used (e.g. as a callback function), the original is removed.

Even if a function can not be inlined (e.g. due to an excessive number of calls to it), it may be possible to make the argument passing more efficient by using registers.

## 3.2 Streamlining

Even though different generations of the x86 processor family all can execute the same code (except when new instructions are used), they prefer it served in different ways. That is something like trying to push elastic objects of one shape through a hole with another shape. It will work, but not as fast as could be if the object had the appropriate cross-section. (This is of course not a big problem if one has the source code available and a compiler capable of processor-specific optimisation/code generation).

One generic way of doing these improvements is to make the tuner capable of as accurately as possible simulate the flow of instructions through execution by having models of how the different processors behave, with emphasis on cycles used (time spent). The models are applied to parts of the code, e.g. one loop at a time and before and after jumps, etc. Then attempts are made to improve the behaviour by applying transformations, like reordering instructions and replace one pattern of instructions by another equivalent one.

For the 386 and 486, the model mainly have to keep track of the cycles required for each instruction. For the Pentium (586), the simulation get more involved by the dual pipeline that is capable of executing pairs of instructions (if they match, i.e. are pairable) [1]. For the 686 there are complications by the possibility of decoding 3 instructions in parallel into $\mu$ops (micro-operations, often more than one per instruction), which are then executed (possibly) out of order at a maximum of 3 per cycle (if they are of the right types) [8], yielding a bumpy road to high speed perfection (because of the difficulty of accurately estimating the actual throughput).

But then it is also straightforward to cope with AMD and other manufacturers' chips too, by creating appropriate models.

This will require tight integration between the editable instruction structures and the actual encodings, so that alignment on word boundaries, instruction length and such things (which also affect execution time) can be taken into account.

## 3.3 Avoiding jumps

Jumps (especially conditional ones) in the code can be devastating for performance. And actually worse so for newer generations of processors, because the execution is done as a long multi-step process (pipeline), which the instructions flow through nicely when they come in an orderly fashion. The processors are capable of following unconditional jumps without much trouble (if they recently followed the same jump, so they know the target). And they are even fairly good at predicting the outcome of a conditional jump and then follow them in the same effortless way. But when prediction fail (i.e. the condition proved to be the opposite), the entire execution pipeline has to be flushed and for out-of-order execution it must be ensured that effects of instructions before the mispredicted jump are reflected in the state of the processor, and no things get done for instructions after the jump.

Getting rid of all conditional jumps is impossible as they *are* the program logic, but some of them may be replaced by conditional moves (which are only

available on recent models ($\geq$ 686), see Listing 3.1) or perhaps some clever arithmetics, avoiding jumping around in the code.

Sometimes it may also be possible to entirely remove some branches (e.g. parts of a switch-statement) if the analysis has determined that the corresponding conditions never can be fulfilled.

| With jump | Without jump |
|---|---|
| ```⋮``` | ```⋮``` |

```
  ⋮                    ⋮
  CMP                  CALC      tmp
  JCC   continue       CMP
  CALC  dest           CMOVNCC   dest,tmp
continue:            label_not_needed:
  ⋮                    ⋮
```

With jump / Without jump

Listing 3.1: Using a conditional move to avoid branching. `cmp` sets the status flags, and may have to be moved to be directly before `cmovncc` if the calculation affect the flags. The correct name is `cmovcc`, `ncc` is used instead of `cc` to show that it would be the opposite condition that has to be tested for. `calc` stand for some calculation. The trick can not be used if the calculation access some memory which the conditional jump is to guard against, e.g. an invalid pointer.

## 3.4 Reordering

As the length of some code traces will have changed during tuning, the entire code will usually have to be relocated to new offsets in the executable. As another tuning operation, the order of functions and even traces should be changed so that code that is close in the call graph also is closely located in the image. This will reduce *paging*. One could also think of moving seldom used branches (like error handling code) into special pages, or use it to fill up places that are bad spots for jump targets because of bad alignment. See Figure 3.1 for an example of straightening out a conditional jump that is almost always taken by replacing it by one that is rarely used instead.

Determining if a branch is taken often or seldom can be done by profiling the program. This is done by instead of tuning it, installing counters after each branch and put the program through test runs.

## 3.5 Rearranging data structures

Data structures usually can not be done much about, but when a structure is exclusively used internally, i.e. not used when communicating with an import library, or read from or written to a file, it can be rearranged.

The most used item of the data structure would then be placed at offset 0, since the encodings for pointers without an offset are shorter. It is also beneficial to have other frequently accessed members at offsets $\leq$ 127, as that allows for the use of an one-byte offset instead of a full 4-byte (32-bit) offset. It is also

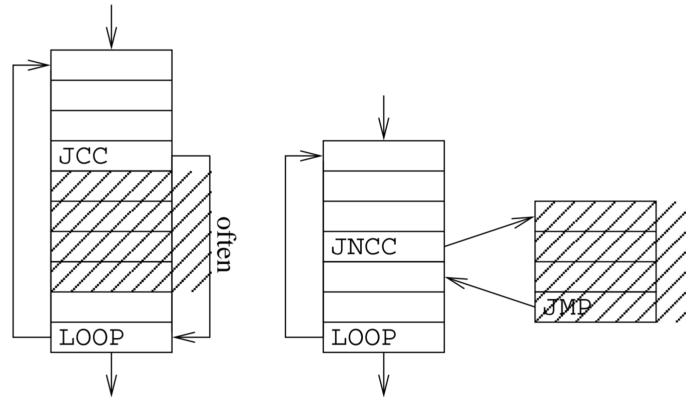



Figure 3.1: Instead of having to jump around the seldom used code (hatched), the normal path is straight forward, and the rarely used code is instead reached via a branch.

possible to use pointers offset by 80h = 128, i.e. pointing inside the structure instead of at the first element, so that the negative one-byte encodings also can be used (making the entire range -128 to 127 available cheaply). Sometimes structures have substructures that are used more often than other members of the larger structure. Then it may be beneficial to have the larger structure being pointed to at the start of the substructure, to avoid an addition when calculating a pointer to the more frequently used sub-structure.

Data items that are accessed close to each other (in the code) should also be neighbours in the data structure, and suitably aligned, to improve the usage of the processor cache.

## 3.6 Going the other way

Some of the above adjustments could also be used to instead aim for the very smallest executable size possible, which would be useful for daemons that are always loaded in memory but not using much CPU time, or for programs running on low end machines with a very limited amount of memory (therefore suffering from heavy paging).

# Chapter 4

# Implementation

This chapter give an overview of the implementation, which is written in C++. The language was chosen for three reasons. The C connection provide easy access to machine-near constructs manipulating bits and bytes (needed to manipulate machine code). The ++ part reduce the risk of the program attaining critical mass[1] and make the program almost write and structure itself. And perhaps most importantly: I know it.

## 4.1 Storage classes

Figure 4.1 show how the packaging file information is stored in a class derived from `executable_image`. The actual code is handled by an `executable` object, and further divided into `call_functions` and `branch_targets`. Each `branch_target` is associated with a trace of instructions as described in Section 2.2.2.

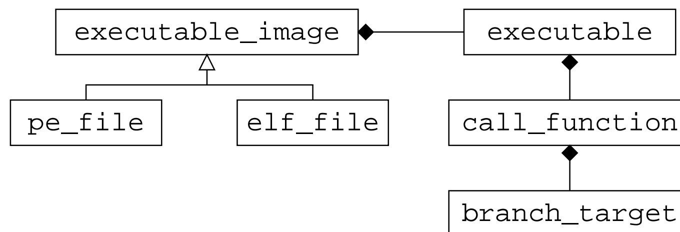



Figure 4.1: Storing an executable.

As shown in Figure 4.2, each assembler instruction is packaged as an `instruction` object. The subclass `branch_instr` keep track of all possible targets of branching instructions ((un)conditional jump, call and return). Each operand is stored separately in an appropriate subclass of `operand`. The instructions are collected into traces by `code_trace` in `branch_target_traces`, while the `branch_target` class store the linkage of each trace to other

[1] When fixing 1 bug introduces $1 + \epsilon$ bugs.

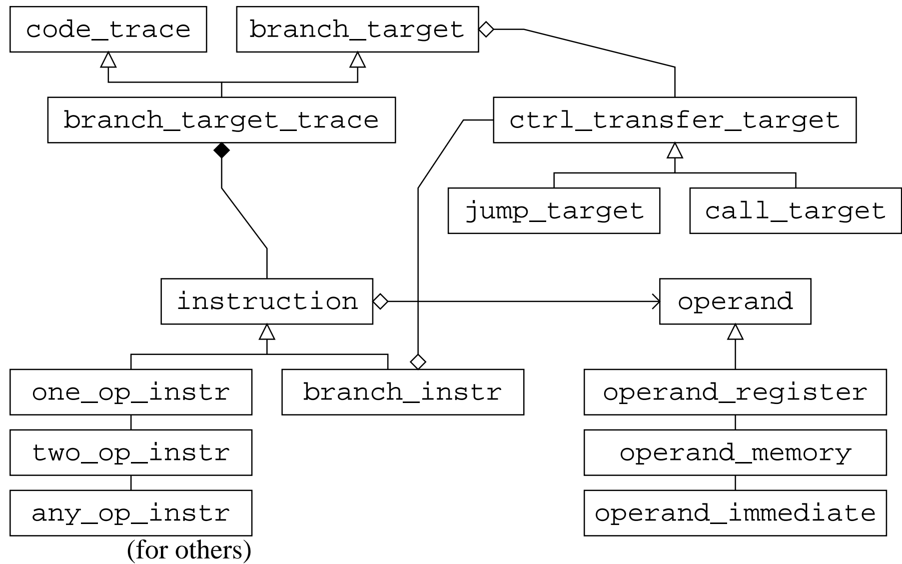



Figure 4.2: Division of the code into functions, traces and instructions.

traces, i.e. a representation of the possible control flow in the program, via subclasses of `ctrl_transfer_targets`.

The data analyser store the calculated changes to the processor (and memory) state after each instruction in `outgoing_state` objects, see Figure 4.3. The known state on entry to a trace or a function is stored in the subclasses `incoming_state` and `entry_state`, respectively. Each `outgoing_state` link to the previous one so that unaffected registers and stack are not forgotten. When the contents of a location (register/stack) is needed, the request is passed backwards through the state objects until the origin is found in an associated `register_state`. If not found, the search is ended at an `entry_state`, which will supply an unknown value if necessary and mark the requested location as an argument to the function being analysed.

Because accesses to memory are of an unordered nature (see Section 2.3.2), the analysed contents of memory other than the stack is stored in a global `memory_state` object.

The value of each location is stored in a subclass of `variable`, depending on what is known about its content. The `variable` object may also have a `variable_content` associated, representing its abstract function, e.g. as a stack pointer.

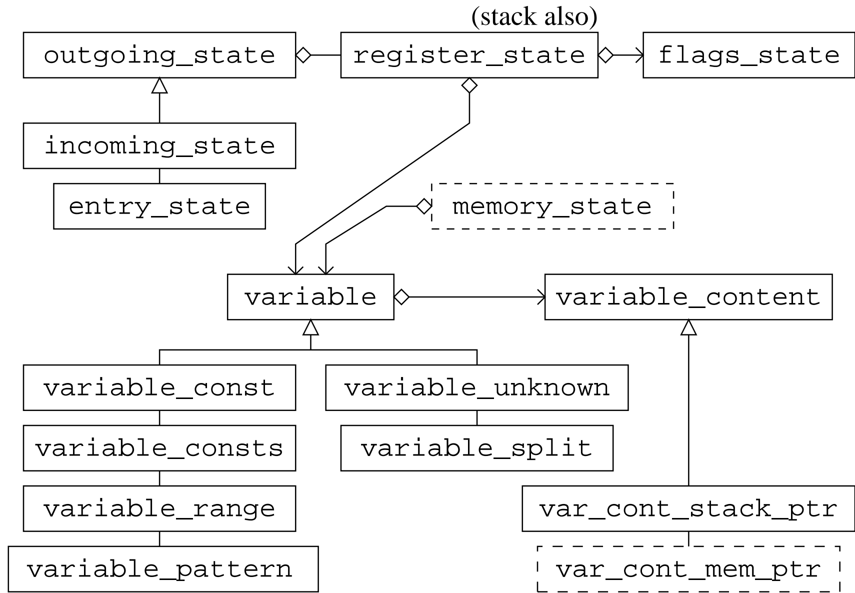



Figure 4.3: Outgoing states from instructions and their variable content.

Figure 4.4 show that the origins of variables are traced via `variable_ops` which keep track of the operands of the instructions that created a variable. The new value of the variable is calculated by member functions of the appropriate subclass for the instruction.

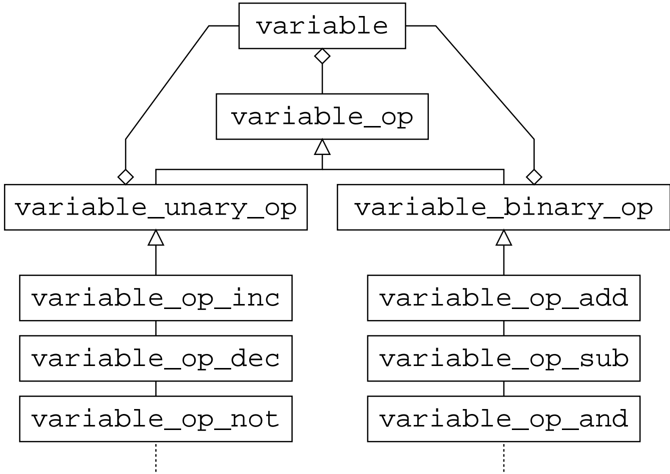



Figure 4.4: Variable sources. Tracking what operations led to a variable.

## 4.2 Working classes

The main parts of the analyser/tuner system are encapsulated in the classes shown in Figure 4.5. A `disassembler` uses an `instruction_decoder` to generate `instructions` from the raw machine code in the executable image and collects them into `call_functions`. Then a `data_analyse_function` investigate the `call_functions`, and a `tuner` will tune the code. An `instruction_encoder` is used to find possible encodings of any `instruction` with the help of a `try_encode_instruction` (for trying different possible encodings).

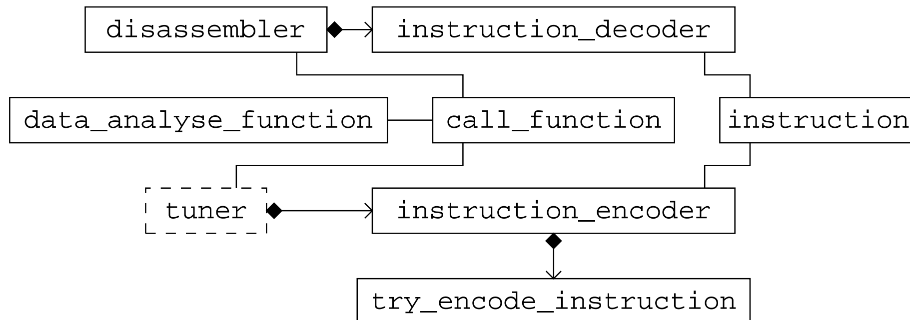



Figure 4.5: The main working classes of the analysing and tuning system, and the objects they operate on.

# Chapter 5

# Results

## 5.1 Implementation

Because of the limited time for this work, only part of the ideas presented here have been implemented so far. Therefore, only some small and academic example programs has been successfully put through the tuner. This is because more than expected small (and large) problems has surfaced, and it take time to drown them in working code. However, no insoluble problems has been found.

## 5.2 Reading and writing executables

The program has successfully read and written very simple PE and ELF files. The PE and ELF files were created manually to keep the amount of executable instructions to a minimum, as the writer is not capable of handling branches very well, and even the simplest compiled C programs bring in some start-up code. The reader/disassembler can handle larger (real) programs.

## 5.3 Changing an executable

A program consisting of two functions (`main` and `func`), has been modified by the tuner to inline `func` into `main`. As the system does not have a functional analyser yet, the data from that was ignored, and inlining was done without any checks for avoiding stack corruption etc. Execution time was reduced from 2.0 s to 0.9 s. This is probably the best improvement this system will ever produce, even when (and if) capable of handling real programs, as the example was chosen to aggressively illustrate the possible benefits of inlining. See Appendix F for details.

# Chapter 6

# Finale

## 6.1 Conclusions

Is this the right way to improve execution speed? While not being the entirely wrong way to make a program better, it is definitely a backwards approach. I would recommend whole program optimising compilation as a way to better utilise the information available in the source of the programs, because much of that has been lost after compilation, and cannot be recovered later. There is no tuning described in this thesis that is impossible for a (whole-program) compiler to do.

And another problem which neither this approach with post-compile adjustments or smarter compilers can treat to any significant extent is reduction of data structures and thus also memory used. Removing unnecessary variables from the program of course also mean that the instructions reading and writing that data are removed. So unless the data structures are reduced as a tradeoff between memory and time consumption, their reduction is generally beneficial. This, however, along with the selection of the algorithms used in a program, is still an order of magnitude more important than any optimising compiler or other automatic tool, and a job for programmers.

## 6.2 Future work

A lot! Quite a few things remain to be done, as outlined below.

### 6.2.1 To do

As described in Section 5.1, the implementation is not in sync with the description in this report and code for much of the described functionality has not been written yet.

There are many things to do before the system has reached a first usable and useful stage, as in a version 1.0. Firstly, the program analyser must be completed and improved so it can cope with most sensibly constructed programs, say 90 % of the basic GNU utilities written in C, in order to be usable. Secondly, the tuner, which does not exist at all yet, must also be able to do most if not all of the mentioned code improvements, to be useful.

Presently, the analyser is lacking much necessary functionality:

- Only the simplest arithmetics instructions and some easy branching is supported (basically the instructions making up example F.1, except the `int`). All other instructions set the analysed processor state to an unknown one[1], making further analysis almost useless, because for example the stack pointer is lost.
- Global memory is not handled at all.
- Data structures and pointers (which are closely related) are not mapped.
- Non-relative branches and function pointers are not handled.

The executable writer is currently only capable of handling simple executables, and will generate garbage for most programs without advising the user of the destruction performed.

It would also be good to expand the range of treatable programs to ones using constructs that effect the execution path in a mildly strange, but reasonably predictable way, like programs using `setjmp` and `longjmp`, and C++ programs using exceptions (which is about the same thing).

The analysed result could perhaps also benefit from the analyser being able to handle a partial ordering of global memory accesses, i.e. when some accesses can be guaranteed to happen before other ones. This can reduce the number of possible values of the memory locations, an example would be some initialisers, which are run on blocks of memory before any other code affect or use it.

Normally, calls to dynamic (shared) import libraries are left as is. But the system could also be made to create a hot version of a program that make heavy use of import libraries by inlining and tuning the used library functions into the executable.

### 6.2.2 User feedback

Using some extra input from the user, it may be possible to allow the analyser to pass some sections of code that it for some reason cannot handle itself. Another kind of feedback is to profile an input program, i.e. measure the usage of different parts of the program's code. The user must then exercise as much as possible of the program, so that most instructions are run at least once, to get a grip in the relative usage of branches.

### 6.2.3 Other uses

Presently, the program can only take input in the form of machine executable code. Expanding it to accept assembler code (which it can output), the program could be used as the assembler backend of a compiler and be the linker at the same time. This way, it would also be possible to propagate some information to the tuner which is otherwise unrecoverably lost at compilation, like `volatile` markings of memory accesses.

[1] As in known to be unknown.

One could also think about using the program as part of a *JIT compiler*. As a JIT compiler normally only deal with part of a program at a time, this would require a tight relationship between the compiler and tuner because the analyser wont be able to see more than a function at a time (or perhaps even less). So when a tuner with assembler transformations and reordering based on a processor model has been implemented, that could be used to relieve the JIT compiler constructor part of the work of creating efficient machine code.

It may also be possible to use the disassembler/analyser to make a decompiler output stage instead of a tuner/executable writer. This would only require that some changes are made to the code graph before it is output to file in source form. Calculational expression trees would have to be converted into readable mathematical expressions, `cmp`/`test`-instructions together with conditional jumps combined into if-statements and while-loops where possible and register and stack usage changed into variables. Anything that can not be transformed to something C-like, would be left as assembler, or macros for each untranslatable instruction.

## 6.3 The source

To facilitate this further work, possibly by someone else, the so far written source code is made available freely (as in freedom), subject to the GNU GPL [9]. The project is called `exmhcg`[2].

[2] `http://www.dd.chalmers.se/~f96hajo/exmhcg/` may, or may not, be of use or available.

# Acknowledgements

I would like to thank my supervisor Håkan Sundell for giving me the opportunity to do this project, for good ideas and inspiring discussions. The encouragement and proofreading of early versions of the manuscript by my brother Henrik is deeply appreciated. I am also grateful to my parents Evy and Lennart for their never-ending support, without which, this work had never been possible. The people of the Subatomic Physics Group are all thanked for their friendship, understanding and for not letting me lose contact with Physics.

# Appendix A

# Motivator

To show the difference between when the compiler had and not had the opportunity to inline an often called function in a program, the computer is instructed to calculate the utterly useless sum $\sum_{i=0}^{N} i + 1$, with $N = 100000000$ so large that execution not is instantaneous and the elapsed time can be measured. The idea is to show the benefit of inlining by placing the calculation of $i + 1$ in a separate function `func`.

```
int main ()
{
  int sum = 0;
  for ( int i  = 0;  i <= 100000000; i++)
    sum += func(i);
  printf  ("%d",sum);
}

int func (int i)
{
  return i  + 1;
}
```

The table below show that when `func` is inlined, i.e. made a part of `main` (which can be done by a compiler if both functions are given in the same source file), the program runs much faster. Both tests were also compiled with the optimisation options `-O3 -fomit-frame-pointer`. Of course, the example was chosen to greatly exaggerate the benefits of inlined code.

| Inline | Compile command | Execution time (400 Mhz Pentium II) |
|---|---|---|
| Yes | `gcc func_and_main.c` | 0.58 s |
| No | `gcc func.c main.c` | 2.31 s |

# Appendix B

# Short introduction to x86 assembly

This introduction is aimed at making the assembler code used in the thesis (especially the appendices) intelligible. A x86 system have eight general purpose 32-bit registers, and an instruction pointer, see Figure B.1. The low 16 bits of the 32-bit registers can be accessed separately, and also as the two lower bytes (8 bits) of the four first registers. The instruction pointer keep track of the address in memory where the currently executed instruction is located.

The processor also have a flags register, which is updated by the arithmetics instructions depending on the result, e.g. zero (or non-zero), overflow and sign. It is then possible to use a conditional jump to branch depending on the outcome of the calculation.

As operands, an instruction can use a register, memory or a constant (also known as an immediate). Of course, a constant can not be the destination. An instruction can also only have one operand being a memory reference, either source or destination, any other must be a register or immediate.

Instructions are written on the form `instr dest,src`. Note the order of the destination and source operands. A memory operand is written as `[pointer]`, where `pointer` is one or two registers, possibly with the second register (usually an index into a data structure) multiplied (scaled) by a small factor (the size of each data item) and an offset: `[base+index*scale+disp]`. Scale is one of 1, 2, 4 or 8, and the offset (displacement) may be either 0, 8 or 32 bits.

Both operands of binary instructions are of the same size, either 8, 16 or 32 bits, with some exceptions for extending smaller types to larger ones.

Usually, the destination operand is overwritten with the result, so if it is preferred to keep both sources, it is necessary to first make a copy of one source:

| Type | C | Assembler |
|---|---|---|
| Overwrite destination | `a += b;` | `ADD EAX,EBX` |
| Keep both | `d = a + b;` | `MOV EDX,EAX` |
| | | `ADD EDX,EBX` |

| Register | Description |
|---|---|
| EAX<br>AX<br>AH \| AL | Accumulator, hard-coded operand for e.g. multiplications. |
| EBX<br>BX<br>BH \| BL | |
| ECX<br>CX<br>CH \| CL | Often used as loop counter. |
| EDX<br>CX<br>DH \| DL | |
| ESP<br>SP | Stack pointer.<br>(SP not used for 32-bit programs.) |
| EBP<br>BP | Base pointer. |
| ESI<br>SI | Source pointer (for string instructions). |
| EDI<br>DI | Destination pointer (for string instructions). |
| EIP<br>IP | Instruction pointer.<br>(IP not used for 32-bit programs.) |

Figure B.1: The general purpose registers of a x86 processor. Apart from EIP, which only can be accessed indirectly, and ESP which always point to the stack, which register is used for something is optional, except when the use of a specific register is the only one possible with some special instructions, such as MUL and DIV or one of several short encodings for operations with EAX.

**Copy values**

First, some instructions for copying data are described. REG is a register, IMM an immediate and [MEM] a memory reference.

| Instruction | Operands | Description |
|---|---|---|
| MOV | REG,REG | Copy a value from the source to the destination. |
| MOV | REG,IMM | |
| MOV | REG,[MEM] | |
| MOV | [MEM],REG | |
| MOV | [MEM],IMM | |
| PUSH | REG | Push the source value onto the stack, i.e. subtract 4 (the normal data size for a 32-bit program) from ESP and write the value to [ESP]. |
| PUSH | IMM | |
| PUSH | [MEM] | |
| POP | REG | Pop a value from the stack to the destination, i.e. read the value from [ESP] and add 4 to ESP. |
| POP | [MEM] | |

### Arithmetics

Next, some instructions for doing integer math and logic are needed.

| Instruction | Operands | Description |
|---|---|---|
| ADD<br>ADD<br>ADD<br>ADD<br>ADD | REG,REG<br>REG,IMM<br>REG,[MEM]<br>[MEM],REG<br>[MEM],IMM | Add the value of the source to the destination. |
| SUB | as above | Subtract source from destination. |
| CMP | as above | Same as SUB, except that only the status flags are updated - the destination is not affected. |
| AND | as above | Bitwise and. |
| OR | as above | Bitwise or. |
| XOR | as above | Bitwise exclusive or. |
| INC<br>INC | REG<br>[MEM] | Add one to the location. |
| DEC | as above | Subtract one from the location. |
| NEG | as above | Negate location. |
| NOT | as above | Bitwise not on location. |

### Branching

Instructions to jump around in the code are also important. Note that when using a relative branch, the offset is added to EIP as it would be when executing the following instruction, not the branching instruction. This mean that a zero offset has no effect (except when being used with a CALL, because the value of EIP is pushed onto the stack).

| Instruction | Operands | Description |
|---|---|---|
| JMP | REL | Relative jump, add the immediate REL to EIP. |
| JMP<br>JMP | REG<br>[MEM] | Absolute jump, set EIP to value of location. |
| CALL | REL | Relative call, push EIP of next instruction onto stack and add the immediate REL to EIP. |
| CALL<br>CALL | REG<br>[MEM] | Absolute jump, push EIP of next instruction onto stack and set EIP to value of location. |
| RET | | Pop the topmost value of the stack into EIP. |
| RET | IMM | Same as above, but also pop IMM bytes of the stack. Used for discarding arguments. |
| JCC | REL | Conditional relative jump. Jumps if the status flags fulfil the condition tested. |
| LOOP | REL | Conditional relative jump. Jumps if ECX is non-zero. |

The CC on the conditional jump stand for a condition code, it can for example be Z or NZ, which would make the program jump if the last calculation produced a result that was zero or non-zero, respectively.

**Some special instructions**

For completeness, the conditional move and interrupt instructions are also mentioned.

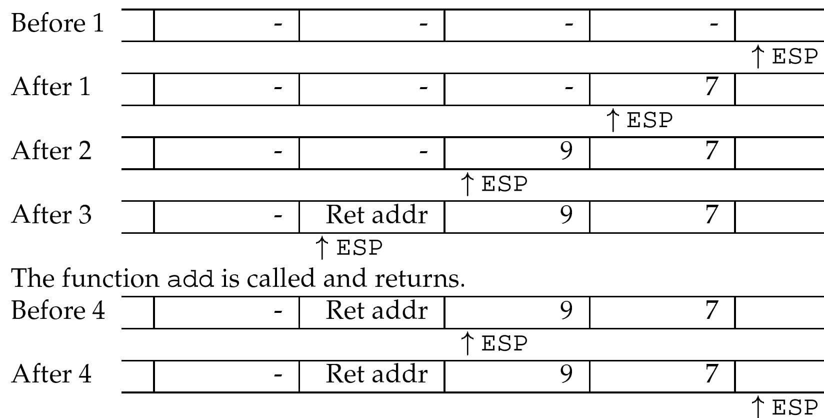
| | |
|---|---|
| `CMOVCC REG,REG`<br>`CMOVCC REG,[MEM]` | Conditional move. The value is only written to the destination if the condition is true. Only available on newer processors ($\geq$ 686). |
| `INT IMM` | Generates an interrupt. This causes the processor to execute the associated interrupt handler. `INT 80` is the system call when running Linux, so control is passed to the kernel. |

# Appendix C

# Execution stack at work

Short illustrations of the stack at work is shown in Listings C.1 and C.2. In the first a function call with two arguments is shown, and the second shows what the stacks looks like to the function called. Note that dealing with 32-bit code, the normal variable size is 4 bytes. A technical detail is that with x86 processors the stack grow downwards, towards lower addresses. This, of course, has implementational consequences, but no conceptual.

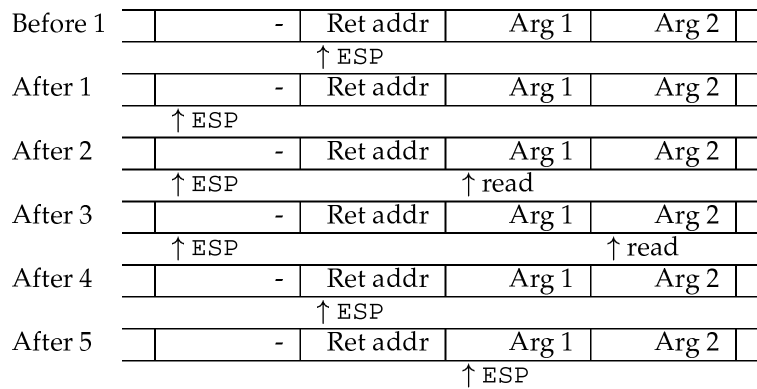
```
1  PUSH   00000007h    ; push argument 2
2  PUSH   00000009h    ; push argument 1
3  CALL   add          ; call the function,
                       ; (result in EAX on return)
4  ADD    ESP,8        ; remove the arguments
                       ; from the stack
```




Listing C.1: A function is called with 2 arguments.

When a function is called, and just before it returns, the stack pointer ESP point to the return address. In listing C.1, this is the memory address of instruction 4.

```
add:
1   SUB   ESP,4          ; grow stack space for one
                         ; local variable (never used
                         ; and uninitialised)
2   MOV   EAX,[ESP+8]    ; get argument 1 (into EAX)
3   ADD   EAX,[ESP+12]   ; add argument 2
4   SUB   ESP,4          ; local variable out of scope
5   RET                  ; (result is in EAX)
```




Listing C.2: A function called with two arguments, and one (unused) local variable.

# Appendix D

# PE and ELF files

Executable files consist of some headers (sometimes located at the end of the file, despite the name) with general information (type and version of machine and operating system to run on, address of entry point for execution etc.) and the location, size and permissions (read, write and/or execute) of the sections that contain the bulk of data. The layout of the headers is specified in [6, 7] and is just packaging. This project is mainly concerned with the contents of the code and data sections.

The main difference between PE and ELF files are the way imports and exports are handled. This is part of the packaging, but has direct consequences for the disassembler/analyser. The other code/data is basically unformatted as it is intended for direct execution and use by the processor.

## D.1 PE file (executable for Windows)

A portable executable (PE) file consist of a relic DOS header from older days, see Figure D.1. After that come the PE header with the necessary loading information (image size, OS version, address of entry point, number of sections). This is followed by section headers, with information about each section in the file (size, location in file and image and permissions).

The data section usually contain an import directory, naming all functions that are to be resolved (provided) by import libraries (dlls) on loading. The actual addresses of these functions (where they happen to be placed in memory when the library is loaded) will be placed at specific locations in a table following the import directory when the program is loaded. Any calls in the code section to the external functions have been encoded to look for the addresses there. Using this information the disassembler can be prevented from trying to find the actual function definitions and instead handle separately such function calls in the code (remembering that is was an imported function).

If the executable image is an import library itself it may also contain an export directory, naming all functions (or variables) that it provides. By reading this information, entry points of functions and locations of variables in the library image are found. Here a problem is encountered: how is it possible to determine if an exported object is a function or variable? By guessing[1]! If the

[1] The fancy word is heuristics.

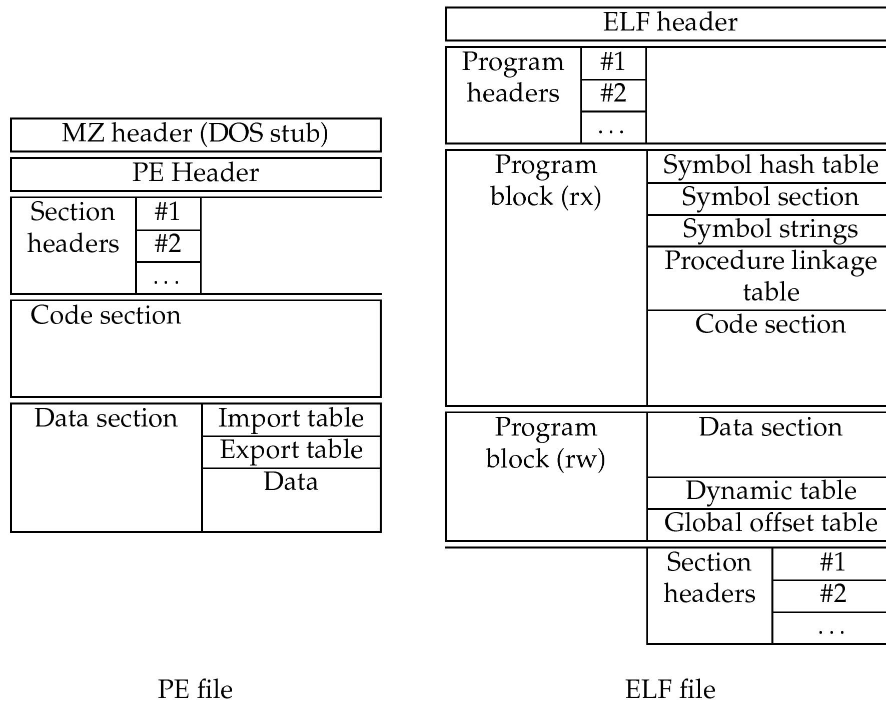



Figure D.1: Structure of the PE and ELF file formats. Both figures are to be read from top to bottom - the division into columns is only to show the structuring of the parts of the files.

address is in the (executable) code section it is assumed to be a function and if not (i.e. in a data section) it is treated as a variable. The only way to validate this assumption is by having some external supply of information. One semi-automatic way could be to parse the include headers for the library. The best (most accurate) solution is to inspect the executables importing the library and actually using the objects in question. It would then be verified that the objects assumed to be functions are called and that objects thought to be variables are made read/write access to, and not the other way around. Also see "Callback functions" under Section 2.3.4.

## D.2 ELF file (executable for GNU/Linux)

An executable and linkable format (ELF) file is also built with an overall header (with address of entry points and number of program blocks and sections), see Figure D.1. The contents of an ELF file has a dual view, it is divided into program blocks for loading into memory, and sections for tools that modify the executable file itself. Headers for the program blocks (which contain block sizes, permissions and locations in memory and file) directly follow the main header, while section headers (with the same information for the sections) are at the end of the executable. The section headers are not required for a file

to be executable, but their absence complicate modifications of the file, as the structuring information of the file must be inferred from other data.

The symbol section, strings and hash make up a combined import/export table. The global offset table contain addresses of imported functions, and the procedure linkage table is a trampoline for external function calls using the addresses in the global offset table. The import/export table of an ELF file does have markers to differentiate between function and data objects (although it may be left unspecified, giving the same identification problem as for PE files).

## D.3 Similarities

So although not exactly the same, PE and ELF files are constructed along the same lines. Not very surprising, as they do the same thing: pack an executable.

However, even if the contents of one format could be (with some slight changes), packaged in the other format, that would not create an usable executable for the other operating system. This is because executables import and use functions from the original operating system and/or use system calls, that do not have direct counterparts on the other system. So an executable file must be packaged in the same format it was read from.

# Appendix E

# Data flow analysis example

Listing E.1 shows an analysed function. Note that the x86 has six flags that are set depending on the outcome of arithmetic instructions. Three extra flags are used by the analyser to be able handle the conditional instructions (mainly jumps) that use combinations of flags (< and > comparisons). This is not necessary for tracking calculations with constants as they can be handled via the real flags. However, comparing two ranges might yield unknown values (both 0 an 1 possible) for the real flags, while some of their combinations are fixed.

| | | | |
|---|---|---|---|
| example: | | `ESP=?/st+0` | `EAX=?` `EBX=?` |
| `PUSH` | `4` | `ESP=?/st-4` | `st-4=4` |
| `PUSH` | `5` | `ESP=?/st-8` | `st-8=5` |
| `ADD` | `EAX,EBX` | `EAX=?` | `fl=?---????????` |
| `POP` | `ECX` | `ESP=?/st-4` | `ECX=5` |
| `ADD` | `ECX,[ESP]` | `ECX=9` | `fl=0---00000100` |
| `ADD` | `EAX,ECX` | `EAX=?` | `fl=?---????????` |
| `ADD` | `ESP,4` | `ESP=?/st+0` | `fl=?---????????` |
| `RET` | | | |

Listing E.1: Instructions to the left and the state change after their execution to the right, except on the first row which show the state on entry and the discovered arguments. `ESP=?/st-4` mean that the actual value of `ESP` is unknown, but that it point four bytes below the stack reference level (stack level on function entry, and remember: the stack grows downwards on a x86). The changes to the status flags are also shown after each modifying instruction.

## E.1 Partial registers

The general purpose 32-bit registers of a x86 processor can be accessed in parts also. E.g. the low 16 bits of `EAX` are called `AX`, and the high and low 8 bits of this are `AH` and `AL`. Listing E.2 show part of a function that exercise these

possibilities. Note that `EBX` is determined to be an input to the function, as the fourth instruction uses the uninitialised high part of it. Also, an unaligned access to the stack is made at the end. Here it can be seen that the machine is *little-endian*, i.e. that the byte order in memory is opposite that of normal writing. The two pushes will write 8 bytes at `st-8`: `78, 56, 34, 12, 98, ba, dc, fe`, and the memory read will get the four middle bytes `34, 12, 98, ba`, giving `ba981234`.

| | |
|---|---|
| example2: | ESP=?/st+0<br>EBX=? |
| MOV EAX,12345678h | EAX=12345678h |
| MOV AL,3 | AL=3 |
| MOV BX,AX | BX=(56h,03h) |
| MOV EDX,EBX | EDX=(?,(56h,03h)) |
| ADD AH,BL | AH=59h<br>fl=0---00000100 |
| MOV ECX,EAX | ECX=(1234h,(59h,03h)) |
| PUSH FEDCBA98h | ESP=?/st-4<br>st-4=fedcba98h |
| MOV CX,[ESP] | CX=ba98h |
| MOV DX,[ESP+02h] | DX=fedch |
| PUSH 12345678h | ESP=?/st-8<br>st-8=12345678h |
| MOV EDX,[ESP+02h] | EDX=(ba98h,1234h) |

Listing E.2: Exercising partial register reads and writes. When a location is used that was not set with the same number of bits, the contents are merged into a split variable, with a low and high part, possibly recursive. This is represented by (high part,low part).

# Appendix F

# Tuned examples

To avoid having to deal with a compiler's start-up code, the example of Appendix A was assembled manually into an ELF file, and put through the tuner which inlined the function call, see Listing F.1.

| Before | | | After | | |
|---|---|---|---|---|---|
| main: | | | main: | | |
| | MOV | ECX,00000000h | | MOV | ECX,00000000h |
| | MOV | EBX,00000000h | | MOV | EBX,00000000h |
| loop: | | | loop: | | |
| | **CALL** | **00000013h [func]** | | **MOV** | **EAX,ECX** |
| | | | | **INC** | **EAX** |
| | ADD | EBX,EAX | | ADD | EBX,EAX |
| | INC | ECX | | INC | ECX |
| | CMP | ECX,05F5E100h[1] | | CMP | ECX,05F5E100h |
| | JNZ/NE | F0h [loop] | | JNZ/NE | **F2h** [loop] |
| | MOV | EAX,00000001h | | MOV | EAX,00000001h |
| | INT | 80h[2] | | INT | 80h |
| | RET | | | RET | |
| func: | | | | | |
| | **MOV** | **EAX,ECX** | | | |
| | **INC** | **EAX** | | | |
| | **RET** | | | | |

Listing F.1: A function call which is inlined by the tuner. The original version executes in 2.0 s, while the inlined need 0.9 s to complete (on a 400 Mhz Pentium II).

[1] 05F5E100h is hexadecimal for 100000000.

[2] `INT 80h` with `EAX=1`, is the system call to exit a program, and `EBX` hold the exit code. So the following `RET` is never executed, but is currently needed because the disassembler/analyser does not realise that.

## Tracking the stack pointer

Stack pointers, `[ESP+n]`, must be adjusted when they point to arguments to account for the return address not being pushed onto the stack when a function is inlined. Other stack pointers (for local variables) are not changed, see Listing F.2. Note that for example the two additions to the stack pointer at the end of the inlined function could be merged into one after inlining, and that the parameter passing on the stack can be avoided/simplified.

| Before | After |
|---|---|
| ⋮ | ⋮ |
| `loop:` | `loop:` |
| `PUSH ECX` | `PUSH ECX` |
| **`CALL 00000014h [func]`** | **`PUSH 00000001h`** |
| | **`PUSH 00000002h`** |
| | **`MOV EDX,[ESP]`** |
| | **`MOV EDX,[ESP+4]`** |
| | ***`MOV EAX,[ESP+8]`*** |
| | **`INC EAX`** |
| | **`ADD ESP,00000008h`** |
| `ADD ESP,00000004h` | `ADD ESP,00000004h` |
| `ADD EBX,EAX` | `ADD EBX,EAX` |
| ⋮ | ⋮ |
| `func:` | |
| **`PUSH 00000001h`** | |
| **`PUSH 00000002h`** | |
| **`MOV EDX,[ESP]`** | |
| **`MOV EDX,[ESP+4]`** | |
| ***`MOV EAX,[ESP+12]`*** | |
| **`INC EAX`** | |
| **`ADD ESP,00000008h`** | |
| **`RET`** | |

Listing F.2: A function call which is inlined by the tuner. Any references to arguments passed (the lines in italics) on the stack are changed to account for the return address never being pushed (by the `CALL` which is removed).

# Bibliography


[1] Michael L. Schmit, *Pentium processor optimization tools*, AP Professional, cop., Boston, 1995.

[2] Tom Shanley, *Pentium Pro and Pentium II system architecture*, Addison-Wesley, Reading, Mass., 1997.

[3] Intel Corporation, *IA-32 Intel® Architecture Software Developer's Manual*, `http://www.intel.com/design/PentiumII/manuals/`, 2002.

[4] Christian Ludloff, `http://www.sandpile.org/`, 2002.

[5] Konstantin Boldyshev, `http://linuxassembly.org/`, 2002.

[6] Tool Interface Standard (TIS), *Formats Specification for Windows™ Version 1.0*, TIS Committee, 1993.

[7] Tool Interface Standards (TIS), *Executable and Linkable Format (ELF), Portable Formats Specification Version 1.1*.

[8] Agner Fog, *How to optimize for the Pentium family of microprocessors*, `http://www.agner.org/assem/pentopt.zip`, 2000.

[9] Free Software Foundation, *GNU General Public License*, `http://www.gnu.org/copyleft/gpl.html`, 1991.

# Glossary

**Alias problem** Several pointers may point to the same object in memory, without any of the pointers (or the code handling the pointer) knowing about the other ones. Then one pointer may affect a memory location without the other ones knowing about it. It is then essential for the other pointers to re-read the value from memory, and not use an old local copy. (Compare the use of `volatile` in C, which does not solve the problem, but deals with the consequences, by forcing a dereferenced pointer to always be re-read, as it may have changed.)

**Alignment** Data should usually be aligned in memory so that its address is a multiple of its size. Access to unaligned data is slower (requires more memory reads and rearrangement of the individual bytes read from memory). Because assembler instructions are variable length, most instructions are not aligned, but execution speed may benefit from having jump/call targets aligned on some boundary, usually 16 bytes.

**Assembler** Human readable form of machine code. `mov eax,-25` instead of `b8 e7 ff ff ff`.

**Callback function** A function C is used as a callback function when its address is sent as a parameter to the function F, and the function F then calls C via that pointer. Callback functions are for example used by search and sort routines (like `bsearch` and `qsort`) to define the relative order between elements, and as filters when listing files in directories (called once for each entry).

**Compiler** A program source file is translated into machine code by a compiler. The result is an object file.

**DMA** Direct memory access. Communication with hardware devices (e.g. graphic cards, sound cards) may be performed as ordinary memory reads and writes, but using addresses in a range not used by the main memory. The memory accesses usually affect the internal state of the device, so that the next read to the same address may give a different answer, even if no write was performed in between.

**Header file** A file used when compiling a program, that hold prototypes (not definitions = the actual code) of external functions, i.e. their name, parameters and return type.

**Inlining** A function is inlined when the call to the function is replaced by the actual definition. It is like when $b(y) = y + 3$ is generated from the statements that $b(y) = a(y) - 2$ and $a(x) = 5 + x$, avoiding the call to $a(x)$. This can be done by the compiler only if it has access to the definition of the inlined function at the same time as it would generate the function call, i.e. the two functions are in the same source file, and is usually done for small functions.

**Linker** One or several object files are merged into an executable file by a linker.

**Little-endian** A format for storing values (of more than one byte) in memory, with the least significant byte at the lowest address, i.e. in the opposite order than it would be written. The x86 architecture is little-endian, some others are big-endian.

**Paging** The process of swapping parts of memory between RAM and the hard drive. (Done at the discretion of the OS to emulate more memory than physically available, but can be using a considerable fraction of CPU time on a machine with too little memory. Ever heard the VM (virtual memory) eating its way through your hard drive?)

**Peep-hole transformation** Improving code by looking at and modifying a little piece (window) of code at a time. I.e. aiming for local optimum.

**Pipeline** Several instructions can be worked on at the same time by the processor, but in different stages of execution by using a pipeline. Execution can be divided into decoding the instruction, calculating a possibly needed memory address, reading memory if needed, doing the actual calculation and writing the result to any memory destination).

**Register-stack operation** Moving a value between a register and the stack. Registers are few, but accessed fast. The stack can be arbitrarily large (within available memory), but operations on it are slower. A value not needed right now can be temporarily stored on the stack, freeing up a register.

**Relocation** When an executable image cannot be loaded at the address that was intended, all absolute addresses must be adjusted by the offset by which the image is moved to the actually used address.

**Stack** A stack is a collection of last in first out objects, usually represented as a list. The basic operations are to push an object onto the top of the stack, and to pop the topmost object off the stack. Think of a pile of plates, which is normally only accessed from the top.