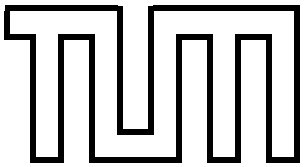
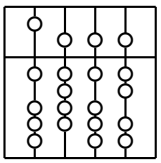




Technische Universität München
Institut für Informatik

Diplomarbeit

Dynamic Re-compilation of Binary RISC Code for CISC Architectures

Verfasser: Michael Steil
Betreuer: Dr. Georg Acher
Aufgabensteller: Prof. Dr. A.Bode

Abgabetermin: 15. September 2004

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.09.2004

Abstract

This thesis describes a dynamic binary translation system that has the following features:

- RISC (PowerPC) machine code is translated into CISC (i386) code. Special problems caused by this combination, such as register allocation and condition code conversion, are addressed.
- A modified hotspot method is used to do recompilation in two translation passes in order to gain speed.
- The first translation pass is optimized for maximum translation speed and designed to be only slightly slower than one interpretive run.
- The system optimizes common cases found in compiled user mode code, such as certain stack operations.

The idea of this system is not to develop a high-level decompiler/compiler combination, but to do translation on a very low level and on an instruction basis.
The recompiler that has been developed contains

- recompiler technology that only needs about 25% more time per instruction for translation compared to a single interpretation, and produces code that is only 3 to 4 times slower than native code.
- recompiler technology that only needs about double the time per instruction for translation compared to a single interpretation, and produces code that only takes twice as long for execution as native code.
- recompiler technology that, without using intermediate code, optimizes full functions and allocates registers dynamically; and achieves speed close to that of native code.

I would like to thank Georg Acher (supervision), Christian Hessmann (discussions, proofreading), Melissa Mears (discussions, proofreading), Axel Auweter (discussions, support code). Daniel Lehmann (discussions, proofreading), Tobias Bratfisch (support code), Sebastian Biallas (discussions), Costis (discussions), Alexander Mayer (proofreading) and Wolfgang Zehtner (proofreading) for their support.

Contents

List of Figures

List of Tables

Chapter 1

Motivation

”Link Arms, Don’t Make Them” - *Commodore 128 Design Team*

There has always been and there will always be different and incompatible computer systems. Although the home computer wars of the 1980s and early 1990s with their extreme variety of platforms are over, there are still a handful of desktop and server architectures today, with different and incompatible CPUs. Whenever there is incompatibility, there are solutions to make platforms compatible:

- **Commodore 128**: Two CPUs made this computer compatible with the Commodore 64 as well as the CP/M operating system.
- **Wine**: This solution makes it possible to run Windows applications on Linux.
- **VirtualPC/Mac**: This application emulates a complete IBM PC on an Apple Macintosh.
- **Java**: Applications written for the Java Virtual Machine can run on many otherwise incompatible computers.

Interoperability can be achieved through hardware or software. The software solution either means porting the source code, or, if no source code is available, emulation.
Regarding the purpose of emulation, most of the available emulators fall into one of the following categories:

1. **Commercial emulators**, mostly intended for migration: These emulators are typically highly optimized and target a machine that is superior to the emulated machine (DR Emulator [1]) or even constructed for optimal emulation performance (Crusoe [6]). Therefore, they avoid problems of the general case.
2. **Hobby emulators**, mostly of classic systems (UAE [14]): Most of these emulators target the i386 CPU, but they are hardly optimized for performance, as sufficient emulation speed is easily achieved because the emulated system is typically clearly inferior.

3. **Portability solutions** (Java [10], .NET [11]): These emulators are highly optimized. Their input language is an artificial intermediate language that includes a lot of meta-data and makes analysis easy. They are more similar to a back end of a compiler than to a conventional emulator.

This thesis is supposed to close the gap between 1 and 2: Commercial emulators often achieve very high speeds, which is easy because they target a superior CPU architecture. Hobby emulators sometimes target an inferior CPU architecture, but they rarely have a need to be fast.
The objective of this thesis is to develop concepts for a recompiler of a RISC architecture that targets a CISC CPU: PowerPC machine code is supposed to be translated into i386 machine code. Both are architectures that are widely used today, and the newest members of each family achieve a comparable performance. So PowerPC to i386 recompilation means translating code to an equally fast system that has an inferior ISA.
The RISC/CISC combination is not supposed to be the single characteristic that distinguishes this work from other recompilers. These four concepts are the basic ideas of what shall be developed:

1. **RISC to CISC**: The recompiler will be designed to translate code to an inferior instruction set architecture. It must deal with differences such as the number of registers and three-register-logic as well as with complex instruction encoding. Low-level solutions are supposed to be found to produce very optimized code.

2. **Recompiler/Recompiler Hotspot Method**: Dynamic recompilers almost always use the hotspot system that combines recompilation with interpretation. In the scope of this thesis, a hotspot recompiler will be developed that never interprets but has two levels of recompilation.

3. **Simple and Fast Recompiler**: Most recompilers are either optimized for simplicity or for producing excellent code. The first pass of this hotspot system will consist of a very fast recompiler optimized for high speed dispatching, decoding and code generation, which still produces fairly good code.

4. **Compiled User Mode Code**: Many recompilers emulate complete computers including the operating system. This solution will be designed for user mode code only, and optimized for code that has been produced by a compiler.

Although some of the ideas involve optimization and speed, the important point is to evaluate algorithms that promise good performance instead of optimizing implementations.
The solutions developed in this thesis can be useful for different applications:

- **Softpear**: Softpear [28] is an Open Source project working on a solution to run Mac OS X applications on i386 CPUs. This will be achieved by using the i386 version of the base operating system Darwin and emulating user mode applications.

- **Software Controlled Radio**: An addition to Digital Audio Broadcasting being researched transmits audio codecs in a RISC-like intermediate language. The RISC to i386 recompiler could be used to target the i386 CPU.

- Hardware Based Recompilation: The simple recompilation method developed in this thesis could be used to construct a PowerPC to i386 recompiler that is implemented in hardware.

Additionally, the source code or the ideas might be used in other PowerPC to i386 emulators, like PearPC [15] or a Nintendo GameCube emulator.
Apart from concrete applications, recompilation and especially RISC to CISC recompilation has some value to research. Although CISC CPUs have been written off more than a decade ago, the majority of desktop computers is driven by i386 CISC CPUs, and the AMD64/EM64T extensions will lengthen the life of CISC even more. This is caused by the fact that there is a gigantic base of software available for this platform, which would be lost when switching CPU architectures. Recompilers can solve this problems; they can help leaving the i386 platform, so recompiler research is important.
But as long as CISC is so important, it is also interesting as a target for recompilation. Research seems to always have concentrated on migration from CISC to RISC, but interoperability between the two is also an important goal.

This thesis is divided into six chapters. The following chapter gives an overview of the technologies that form the basis of the work on this topic: recompilation in general and the i386 and PowerPC CPUs. Chapter three describes the basic concept as well as all details of the design of the recompiler that has been developed. Interesting implementation details are described in chapter four. Chapter five evaluates the code quality as well as the speed of the generated code. Limitations of the current design and ideas for improvements are discussed in chapter six. The Appendix contains speed measurement code as well as detailed statistics that are referred to throughout this thesis.

Chapter 2

Introduction to Emulation, RISC and CISC

2.1 Introduction to Emulation and Recompilation

Emulation is the technique that allows running software that has been designed for machine A on an incompatible machine B. This is achieved by simulating on machine B the environment that the application would expect on machine A. This can include the emulation of CPUs, peripheral hardware, as well as software interfaces. There are many different uses for emulation:

- **Historical Preservation**: Old machines often have some historic value, and there might be only few or even no working machines left. Emulation makes sure that these systems can still be explored and learnt from today. Emulators are available for many classic machines, like the DEC PDP series and the Altair 8800 [12].

- **Migration**: Work flow in a company may for decades depend on an application that is only available for a single architecture. This application may not have been ported to other (later) computer systems, because the original producer has gone out of business, the source code of the application has been lost, or because the application has been implemented in assembly language. If the old architecture ceases to exist and the old machines stop working, emulation makes it possible to continue running the application on current machines, maybe even faster than on the old machine. In general, migrating from one architecture to another can be simplified by emulation, because the user can have all the benefits of the new system without sacrificing compatibility with the old system.

  Apple's DR Emulator [1] is one of the most prominent examples of this kind of emulators. It allowed easy migration from Motorola M68K based computers to PowerPC machines, by translating old code and even parts of the operating system on the fly. Intel provides an emulator of i386 (IA-32) Windows applications for IA-64 CPUs.

- **Interoperability**: Instead of making new machines compatible to older ones, emulators can also be used to make different modern systems compatible with each other.

This can be done by either making the environment of one system available on another system, or by creating a new system that is somewhat close to several existing systems, and writing emulators of this artificial system for physical systems. The Java Virtual Machine is such an artificial machine: A Java runtime environment is an emulator for a system that does not exist. The Java VM has not been designed to be implemented in hardware, but to be emulated on common desktop and server architectures. Examples of emulators of one current system on another one for the sake of interoperability are VirtualPC/Mac (IBM PC on Apple Macintosh), Digital FX!32 (i386/IA-32 Windows applications on Alpha) and WINE (Windows on Linux, actually only an API converter)[1].

- **Software Development**: Developing and testing operating systems, OS components and system services can, on some platforms, easily crash the test machine. Emulating the target system makes sure that only the emulated system crashes, that the emulated system can be restarted quickly and that the development computer remains unaffected. The emulator might even make debugging easier by providing additional crash information that would not be available on a physical machine, or by allowing single-stepping through an operating system.

  If applications are developed for a very small and slow device, it is often cross-developed on a powerful machine and loaded into the real hardware for testing. If the transfer to the physical device is slow or memory would not suffice for debug builds, the software can be tested in an emulator of an enhanced version of the device, running on the development machine.

  Bochs is a very popular IBM PC emulator that runs on many host platforms. It is currently too slow to be used for interoperability, but it is a valuable tool for operating system development. VMware is a similar product that provides higher speed but less accuracy. AMD distributed an x86-64 (AMD64) emulator for operating system development before shipping the system in hardware. Microsoft includes an emulator of Windows CE devices with their Windows CE cross-development environment.

- **Hardware Development**: When developing new hardware, especially a CPU, it is generally a bad idea to complete the implementation in hardware before testing any real-world applications on it. Emulators can help verifying the efficiency of both the instruction set and the actual implementation in silicon.

Most well-known emulators fall into the ”migration” and ”interoperability” categories: Emulators exist for many classic computer systems and gaming consoles since the 1970s up to the current generation, mainly for providing classic games or games intended for recent gaming consoles on modern desktop computers. While these are often hobby projects (Basilisk II, UAE, VICE, SNES9X, Nemu, Dolwin, cxbx), compatibility solutions between current systems are typically commercial software (VirtualPC, VMware, Win4Lin).

[1] It is a matter of perspective whether to regard certain emulators as falling into the migration or interoperability categories.

2.1.1 CPU Emulation Accuracy

Depending on the exact purpose of the emulator, it need not provide maximum accuracy for the intended applications to run flawlessly. Less accuracy can result in less development time or higher emulation speed. Emulators used for development generally need to be more accurate than emulators for migration.
[44] defines the following five accuracy levels:

- Data Path Accuracy is the highest level of accuracy. It simulates all internals of the hardware exactly, including the data flow inside the CPU. This kind of emulators represents the software version of hardware implementations of a system. This level of accuracy is mostly needed for hardware development, i.e. the emulator is a prototype of the actual device.

- Cycle Accuracy means emulating the hardware including the ”cycle exact” timing of all components. This type of emulators only implements the exact specification of hardware, but not its implementation. Many 1980s computer games (typically on 8 and 16 bit computers or gaming consoles) require this level of accuracy: Because it was easier to count on the timing of, for instance, CPU instructions as it is today, games could easily use timing-critical side effects of the machine for certain effects. For development of real time operating systems, cycle accurate emulators are needed as well. For example, VICE [13] is an emulator that emulates a Commodore 64 and its hardware cycle-accurately.

- Instruction Level Accuracy of an emulator includes emulating every instruction of the CPU including all its side effects, but ignoring its timing. Modern computer systems (not counting embedded real time systems) can come with one of many differently implemented, but compatible CPUs with very different timing, so modern applications cannot rely on exact timing. The Virtio ”Virtual Platform” [4] emulators, which are meant for operating system development, provide instruction level accuracy for emulation of various embedded CPUs.

- Basic-Block Accuracy does not include the execution of individual instructions. Instead, blocks of instructions are emulated as a whole, typically by translating them into an equivalent sequence of native instructions. This level of accuracy does not allow single stepping through the code, and advanced tricks like self modification do not work without dedicated support for them. The Amiga emulator UAE [14] is an example of such an emulator: Motorola M68K code is translated block-wise into i386 code.

- High-Level/Very-High-Level Accuracy implies techniques to detect certain sequences of code, typically library routines, that get replaced (not: translated) with equivalent native code. These sequences do not necessarily the exact same work as the original, although the return values must be identical. Instead of emulating hardware registers, many emulators of recent gaming consoles detect library code to access video and audio hardware, and send the requests to the host hardware without executing the original library code. The Nintendo 64 (Ultra 64) emulator UltraHLE [19] was

the first widely-known emulator to apply this technique. Some emulators of computer systems, such as IBM PC Emulators VirtualPC [2] and VMware, as well as the Amiga emulator UAE can optionally do high-level emulation, if the user installs a set of device drivers that directly divert all requests to the host machine without going through hardware emulation.

The lower the level of emulation, the more possible uses there are, but also the more computing power is needed. A data path accurate emulator, if implemented correctly, will without exception run all software that has been designed for this system, but the speed of the host machine must be many thousand times faster than the speed of the emulated system in order to achieve its original speed. Emulators that implement high-level or very-high-level accuracy cannot be used for all these purposes, and often even only run a small subset of all existing applications, but they can achieve realtime speed on machines that are only marginally faster.
For example, for realtime emulation of a Commodore 64 system with its two 1 MHz 8 bit CPUs, a 500 MHz 32 bit CPU is required (VICE emulator, cycle accuracy), but the same host computer would also run many games of the 93.75 MHz 32/64 bit Nintendo 64 (UltraHLE, high-level accuracy).

2.1.2 System Emulation vs. User Mode/API Emulation

High-level accuracy is a technique made possible by the fact that most current systems are layered: While old software (e.g. home computer and MS-DOS software, 8 bit games) was often monolithic and contained all code from low-level hardware access up to the application's logic, current systems leave multitasking, memory management, hardware access, I/O and presentation logic to the operating system and the system libraries.
So if modern computer systems are supposed to be emulated, there are two approaches to simulate the environment of an application:

- Emulate the complete system including all hardware. The original operating system will run on top of the emulator, and the applications will run on top of this operating system.

- Emulate the CPU only and provide the interface between applications and the operating system. Operating system calls will be high-level emulated, and the applications run directly on top of the emulator.

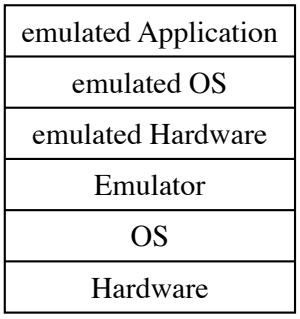
There are two special cases about the CPU emulation: If the CPU is the same on the host and the emulated machine, it need not be emulated. Also, if the source code of the application is available, CPU emulation may not be necessary, as the application can be compiled for the CPU of the host computer. In both cases, only the rest of the machine, i.e. either the machine's support hardware[2] or the operating system's API has to be provided to the application.

[2] In the following paragraphs, this support hardware, which includes devices like video, audio and generic I/O, will just be called "hardware".

Table 2.1: Possible solutions for system emulation

	same API	diff API
same CPU	-	VM (VMware)
diff CPU	VM (PearPC)	VM (VirtualPC/Mac)

Figure 2.1: Layers of System Emulation



There is also a special case about hardware emulation: If the operating system the application runs on is available for the host platform, i.e. only the CPUs are different, it is enough to emulate the CPU and hand all operating system calls down to the native operating system of the host.

2.1.2.1 System Emulation

Table 2.1 summarizes what has to be done in different cases if the whole machine is supposed to be emulated, with the original operating system code running in the emulator:
In case the CPU is the same, but the API is different, the CPU has to be virtualized, and the supporting hardware has to be emulated. VMware [3] is an example of an emulator that runs a PC operating system on top of another PC operating system, without CPU emulation. The Mac OS X Classic Environment (”Blue Box”) runs Mac OS 9 on top of Mac OS X by providing the old operating system with its simulated hardware environment - again without CPU emulation.
If both the CPU and the API are different, the complete hardware, including the CPU, has to be emulated. VirtualPC/Mac is an example of a PC emulator on Macintosh, typically used for running i386 Windows applications on PowerPC MacOS X.
The situation is no different if the operating system APIs are the same and only the CPUs differ, since the operating system API of the emulated machine only exists in the emulated world: PearPC [15] is a PowerPC Macintosh emulator for i386 Linux and i386 Windows - it does not matter whether it runs PowerPC Linux inside i386 Linux, or PowerPC MacOS X inside i386 Windows: The emulator is not aware of the emulated operating system.

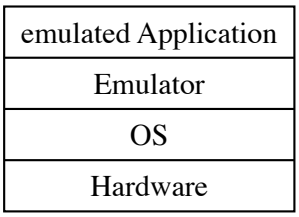
2.1.2.2 User Mode & API Emulation

Concepts that do not emulate the hardware but provide the application with the operating system API are summarized in table 2.2. In case the CPU is the same and the API is differ-

Table 2.2: Possible solutions for user mode and API emulation

	same API	diff API
same CPU	-	API conv. (Wine)
diff CPU	CPU emul. (FX!32)	CPU emul. & API conv. (Darwine)

Figure 2.2: Layers of User Mode Emulation



ent, the code can be run natively. Only the library and kernel calls have to be trapped and translated for the host operating system. Wine [22] is a solution that allows running i386 Windows executables on various i386 Unix systems. It consists of a loader for windows executables, as well as reimplementations of Windows operating system and library code, most of which make heavy use of the underlying Unix libraries.
If both the CPU and the API are different, the CPU is emulated and the API has to be simulated as in the previous case. Darwine [23] is a version of Wine that is supposed to allow i386 Windows applications to run on PowerPC Mac OS X.
If an operating system is available for multiple incompatible CPU types, it is possible to create an emulator that only emulates the CPU, and passes all system and library calls directly to the host operating system. DEC/Compaq FX!32 [7] is a compatibility layer that allows running i386 Windows applications on the Windows version for Alpha CPUs. A similar solution, called the ”IA-32 Execution Layer”, is provided by Intel to allow i386 Windows applications run on Windows for IA-64[3] [9].

2.1.2.3 Advantages of the Respective Concepts

Both concepts have their respective advantages and disadvantages. Emulation of the complete hardware is operating system agnostic, i.e. when the emulation is sufficiently complete, it can run any operating system, which includes many different and incompatible operating systems, as well as future versions of an existing operating system - it can even run applications that bypass the operating system completely.
Also, higher compatibility is easier to achieve, as the interface between the operating system and the hardware is typically less complex than the interface between applications and the operating system, and full compatibility with an operating system and all of its applications can be tested more easily: If the operating system works flawlessly in the emulator, most applications that run on top of this operating system are very likely to work as well. There is no such test for CPU/API emulators, not even if many applications run flawlessly, other applications may use a system function that is not implemented correctly and will

[3] IA-64 CPUs include a unit for executing i386 code in hardware, but recompilation can be faster.

thus crash the application.
On the other hand, CPU/API emulation also has its advantages, the most important one being effective speed. Since these emulators do not have to support system timers, CPU interrupts and virtual memory, CPU emulation is a lot easier and thus faster. For example, interrupts were major problem in the development of the Transmeta Crusoe CPU, which internally translates i386 code into its own language: As all code is recompiled, the current state of the CPU has to be discarded if an interrupt occurs, and the current block has to be interpreted again, so that execution can be interrupted at the correct instruction with the correct state of the i386 CPU [6]. The PowerPC Macintosh system emulator PearPC only reaches a fraction of the host machine's performance inside the emulator, as every single memory access has to go through the emulated MMU (virtual memory) logic, which takes a minimum of 50 clock cycles [55].
Easier CPU emulation is not the only reason for increased speed of a CPU/API emulator. All system and library calls are executed in native code, that is, base functionality of the operating system like memory management (including "memcopy"), encryption and file system decoding do not have to be translated into native code, as they are native already. Also, I/O data has to go through less layers, as the following example shows: On a normal system, if a file is supposed to be written to disk, the (simplified) data flow looks like this:

```
application->API->file system driver->hard disk controller->hard disk
```

On a system emulator, data goes through more layers:

```
application->API->file system driver->simulated hard disk controller->
host API->file system driver->hard disk controller->hard disk
```

It is a lot simpler on a CPU/API emulator:

```
application->simulated API->host API->file system driver->
hard disk controller->hard disk
```

Another advantage of a CPU/API emulator is its integration into the host operating system. A hardware emulator would run as a single application on top of the host operating system, with all emulated applications running inside the emulator, so working with both native and emulated applications can be quite awkward for the user. Emulated and native applications also cannot access the same file system, share the same identity on the network or do interprocess communication across emulation boundaries. But many hardware emulators add additional functionality to circumvent these problems by introducing paths of communication between the two systems. VirtualPC/Mac and the Mac OS Classic Environment, for example, apply a technique to route the information about running applications and their windows to the host operating system. "Classic" applications even appear as separate applications on the Mac OS X desktop. Host file system and network access are typically done by installing special drivers into the emulated OS that connects the emulated and the host interfaces. Of course, again, direct translation of the simulated API to the host API can be a lot faster than these routes across both systems.

Finally, hardware emulators require a complete copy of the emulated operating system, i.e. a license has to be bought. CPU/API emulators do not need this license.
Although the advantages of the CPU/API emulator outnumber those of the hardware emulator, it is not the ideal concept for all applications. There are many examples of emulators based on either of these concepts. For Windows on Linux, there is Wine, but there also is VMware. For Windows on Mac, there is g, but there also is Darwine. And: For Mac on i386 Linux, there is PearPC - but there also is SoftPear.

2.1.3 Interpretation and Recompilation

> ”The most commonly used commercial encryption method is compilation of sources into executables. In this view a CPU can be considered as a decryption device.” - *Frans Faase, 1996*

The heart of every emulator is CPU emulation. Therefore it must be made sure that it is well designed, to achieve maximum accuracy, flexibility or speed, depending on what is needed most. There are two kinds of CPU emulators: interpreters and recompilers.

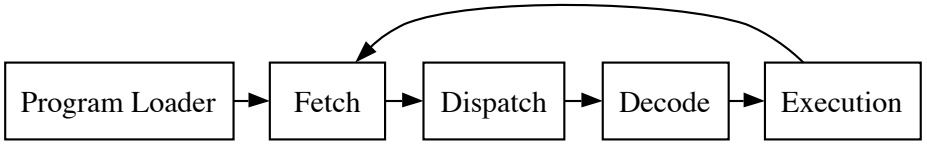
2.1.3.1 Interpretation

An interpreting emulator is the most obvious type of emulator. It executes one instruction after another, as the program runs, i.e. it simulates the instruction cycle of the CPU: The opcode gets fetched, dispatched, operands get decoded, and the instruction gets executed. After that, the interpreter jumps to the beginning of the interpretation loop. The following C code illustrates this:

```
void interpreter(int *code) {
    int *ip = code;
    int instruction, opcode, rs, ra, rb;

    while (1) {
        instruction = *ip;
        opcode = instruction >> 26;
        switch (opcode) {
            case 0:
                rs = (instruction >> 21) & 31;
                ra = (instruction >> 16) & 31;
                rb = (instruction >> 11) & 31;
                register[rb] = register[rs] | register[ra];
            case 1:
                ...
        }
        ip++;
    }
}
```

Figure 2.3: Interpreter Loop



Interpreters need little memory, are easy to write, easy to understand and because of their simplicity, even implementations in a high-level language can reach a speed close to the maximum possible for an interpreter. Typically being written in high-level languages, they are also very portable. They can reach data path, cycle and instruction level accuracy and are therefore mostly used for the verification of a CPU during its development cycle, for cross-development as well as for emulating 8 bit computers and gaming consoles.
Unfortunately, interpreters are inherently slow. The following code is supposed to illustrate the code flow on the host machine (i386) to interpret a single (PowerPC) instruction (”or r5, r5, r3”; note that this code is in AT&T syntax, as described in section 2.3.3):

```
 1 mov (%esi), %eax              // fetch
 2 add $4, %esi
 3 mov %eax, %ebx                // dispatch
 4 and $0xfc000000, %ebx
 5 cmp $0x7c000000, %ebx
 6 jne opcode_1_is_not_31
 7 mov paragrapheax, %ebx
 8 and $0x000007fe, %ebx
 9 jmp *dispatch_table(,%ebx,4)
--------------------------------
10 mov %eax, %ebx                // decode
11 shr $21, %ebx
12 and $31, %ebx
13 mov %eax, %ecx
14 shr $16, %ecx
15 and $31, %ecx
16 mov %eax, %edx
17 shr $11, %edx
18 and $31, %edx
19 mov registers(,%ebx,4), %eax  // execute
20 or  registers(,%edx,4), %eax
21 mov %eax, registers(,%ecx,4)  // writeback
22 jmp interpreter_loop          // loop
```

In this example, a PowerPC ”or” instruction is interpreted. i386 instructions 1 and 2 fetch the 32 bit PowerPC instruction. In instructions 3 to 6, the first opcode field (”31” in this example) gets extracted. Since the most likely value is ”31”, the code is optimized for this case, and the branch in instruction 6 is not taken. Instructions 7 to 9 extract the second

opcode field (”444” in this example) and jump to the implementation of the PowerPC ”or” instruction, using a jump table. The register numbers get extracted in instructions 10 to 18, in instructions 19 to 21, the actual operation is done, and instruction 22 jumps to beginning of the interpreter loop.
This is highly optimized assembly code, and there is probably no way to speed it up significantly[4]. A single source instruction got expanded to 22 destination instructions in this case, and many other source instructions would even require more destination instructions to interpret. But this does not mean that interpretation is about 22 times slower than direct execution - it is even worse.
Modern processors depend on pipelining. While one instruction is being executed, the next one is already decoded, and another one is being fetched. Branches and jumps with an undetermined target are a problem: The instructions after the control flow instruction cannot be fetched or decoded as long as the target of the control flow instruction is unknown. If the target is known early enough or the control flow instruction prefers a certain target (branch prediction), the performance penalty is reduced - but in the case of an interpreter, the branches and jumps (instructions 6 and 9 in the example) do not have an early-known target nor do they prefer one certain target.
An AMD Duron (1200 MHz) for example spends about 13 clock cycles in one iteration of the interpreter loop above, if the indirect jump always hits the same target. If it does not, one iteration takes about 28 clock cycles (see Appendix A). Given the source instruction on its native hardware would normally be completed in about half a clock cycle on average, the interpreter is slower than the original by a factor of more than 50. If an i386 and a PowerPC CPU, both at 1 GHz, have comparable performance, emulating a PowerPC on the i386 using an interpreter would result in an emulated machine that has the performance in the class of a 20 MHz PowerPC.
Measurements [56] done by the QEMU authors indicate that Bochs, an interpretive i386 emulator, is about 270 times slower than native code on the same machine. This additional difference is most likely caused by the fact that CPU flags have to be calculated after every arithmetic or logic function on the i386, and that Bochs has to emulate the MMU as well.

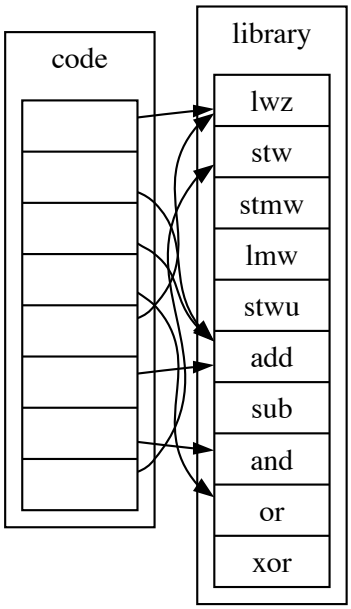
2.1.3.2 Recompilation

> ”The idea of a processor that does Dynamic Binary Translation did not seem very appealing to venture capitalists. That’s when we came up with the term ’Code Morphing Software’ as a marketing tool to get more people interested in funding the Crusoe processor development.” - *David Ditzel, Founder & CEO, Transmeta*

Interpreters are slow because of the dispatcher jumps and because so many instructions are needed for interpreting a single source instruction. These facts have been well-known from interpreters for high-level programming languages as well: Compilers move the decoding of the source language from execution time into compile time, increasing execution speed.

[4] Reordering the instructions might be a good idea, though.

Figure 2.4: Threaded Code



Threaded Code Recompilers using the ”threaded code” [48] [49] concept are the most easy ones to implement, as they are closest to interpreters. An interpreter loop consists of the fetch/dispatch sequence and many short sequences that implement the specific source instructions. A recompiler that makes use of threaded code translates every source instruction into a function call of the implementation of this instruction. Typically the decoding of the instruction is also moved into the recompiler, i.e. the implementations of instructions do not have to do the decoding, but they are called with the already decoded parameters.
The example from above which illustrates the code executed when interpreting in i386 assembly a single PowerPC assembly instruction, changed to a ”threaded code” recompiler, would look like this:

```
  1 mov $3, %ebx
  2 mov $5, %edx
  3 mov $5, %ecx
  4 call library_instruction_or // = 5
-------------------------------------
  5 mov registers(,%ebx,4), %eax
  6 or  registers(,%edx,4), %eax
  7 mov %eax, registers(,%ecx,4)
  8 ret
-------------------------------------
  9 [...]
```

The main function loads the parameters of the source instruction (in this case 3, 5 and 5 for ”or r5, r5, r3”) into registers and calls the function that implements the source instruction

”or” (instructions 5 to 8), which does the work and returns. This example only requires 8 target instructions, compared to 22 in the case of the interpreter, and it eliminates the pipelining problem.
The generated code can be improved easily by inlining the code of the subroutines, instead of calling them:

```
1 mov $3, %ebx
2 mov $5, %edx
3 mov $5, %ecx
4 mov registers(,%ebx,4), %eax
5 or  registers(,%edx,4), %eax
6 mov %eax, registers(,%ecx,4)
```

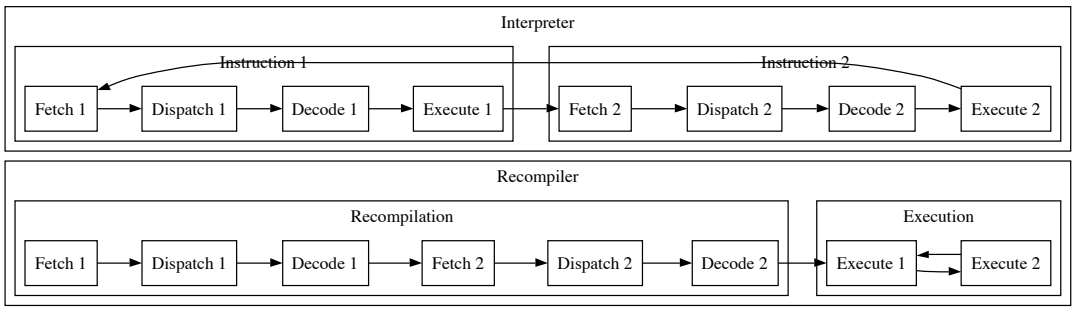
This way, the source instruction is translated into only 6 target instructions, but the generated code is still ”threaded code”, because it is composed of fixed sequences. An AMD Duron 1200 spends about 3 clock cycles on the code above (see Appendix A) — this is 10 times faster than the interpretive version from above, but still about 5 times slower than native code.
QEMU [16] is an example of a recompiler that makes use of threaded code. QEMU has multiple source and multiple target architectures, so the architecture had to be as flexible as possible. The source instructions are implemented as C functions like in an interpreter, and get compiled into machine code during compilation of the emulator. At run time, these sequences, which are then embedded into the emulator’s code, will be put together to form the recompiled code. This architecture allows to add new target architectures easily, and source architectures can be added by adding implementations of all source instructions in C, as it would have to be done for an interpreter. QEMU is about 4 times slower than native code [56].
Some video games have been ”ported” to new platforms by translating them using threaded code: Squaresoft’s Final Fantasy 4 to 6 rereleases for the Sony PlayStation were statically recompiled (see below) versions of the original version for the Nintendo SNES. A rather simple application translated the complete game code from WDC 65816 machine code to MIPS machine code. The same is true for Capcom’s Megaman 1 to 8 on the PlayStation 2 and Nintendo Gamecube, released as ”Megaman Anniversary Collection”: Megaman 1-6 had been for the Nintendo NES, 7 for the Nintendo SNES, and 8 for the PlayStation [66].
All in all, a ”threaded code” recompiler can be implemented with little more effort than an interpreter, but it is radically faster than an interpreter.

Real Recompilation Threaded code is still quite inefficient code. The optimal translation of ”or r5, r5, r3” into i386 code would look like this:

```
1 or  %eax, %ebx
```

Given the i386 register EAX corresponds to the PowerPC register r5, and EBX corresponds to r3 in this context, this i386 code is equivalent to the PowerPC code. This instruction is just as fast as native code - because it is native code.

Figure 2.5: Interpretation vs. Recompilation



A real recompiler does not just put together fixed sequences of target code from a library, but works a lot more like a compiler of a high-level language. There is no common concept that is shared by all recompilers. Some recompilers translate instruction by instruction, others translate the source code into intermediate code, optimize the intermediate code, and finally translate it into target code. The following sections discuss some common recompiler concepts.
Although recompilers are radically faster than interpreters, they typically need more memory. Especially dynamic recompilers (see below) have this problem, as they have to keep both the source and the target code in memory, as well as lots of meta information. Therefore, on small systems (e.g. Java on cellphones), interpretation is often preferred to recompilation.

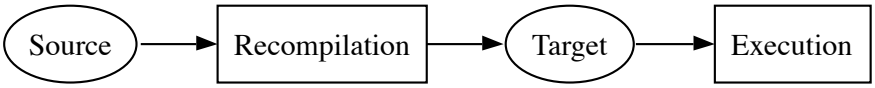
2.1.4 Static Recompilation vs. Dynamic Recompilation

The whole idea of recompilation is to move recurring work out of loops: The work that has to be done for one source instruction gets split into the dispatching/decoding and the execution part, and everything but the execution part gets moved outside of loops, so loops in the source code do not lead to repeated dispatching and decoding of the same instructions. There are two basic possibilities where exactly to move dispatching and decoding: just directly outside of loops (dynamic recompilation), or outside the execution of the complete program (static recompilation).

2.1.4.1 Static Recompilation

Static recompilation translates the complete source code ”ahead of time” (AOT), i.e. before execution, just like a compiler for a high-level language. The resulting data is a standalone executable file that does not include any translation logic. The translated program will be indistinguishable from native programs.
Like standard compilers, static recompilers can do extensive optimization during translation, as static recompilation can be done completely offline and is therefore not time critical. Since all code is known at translation time, global optimization is possible, i.e. complete functions can be analyzed for example, and it is immediately known where a function is called from. Because of all this, static recompilation can generate very fast output code.

Figure 2.6: Static Recompilation



But static recompilation also has some severe problems. The translation from high-level source code to machine code resulted in the loss of some data. Recompilers have to reconstruct a lot of information from the machine code, but some information may be impossible to reconstruct. Distinguishing between code and data is nontrivial: On a von Neumann machine, code and data coexist in the same address space and can therefore be intermixed. On the ARM platform, machine code often loads 32 bit data constants from the code segment, as PC-relative accesses are handy on the ARM.

```
        [...]
        ldr r0, constant            // ldr r0, [pc, #4]
        ldr r1, r2
        bx  lr
constant:
        .long 12345
```

On most platforms, C switch instructions are translated into jumps using a jump table (PowerPC code):

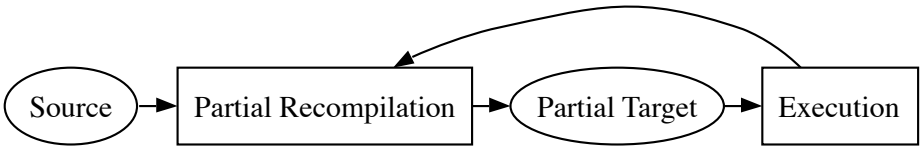
```
        [index to table in r3]
        mfspr   r2,lr           ; save lr
        bcl     20,31,label     ; lr = label
label:  mfspr   r10,lr          ; r10 = label
        mtspr   lr,r2           ; restore lr
        cmplwi  r3,entries
        bgt     default         ; c > entries -> default
        addis   r6,r10,0x0      ; r6 = label
        rlwinm  r0,r3,2,0,29    ; r0 = c << 2
        addi    r5,r6,table-label ; r5 = table
        lwzx    r4,r5,r0        ; get entry from table
        add     r3,r4,r5        ; address of code
        mtspr   ctr,r3          ; ctr = address
        bctr                    ; jump to code
table:
        .long case0 - table
        .long case1 - table
        .long case2 - table
        [...]
case0:
        [...]
case1:
        [...]
```

By just translating instruction by instruction, the recompiler would treat the jump table as machine code. This example also shows something else: A static recompiler cannot easily find out all code in the source program: A tree search through the program (functions are nodes in the tree, calls are edges), starting at the entry point, would not reach the destinations of the jump table in the example above.
Self-modifying code is regarded as a very bad habit that can break a lot - it also makes it impossible for static recompilers to generate a correct translation, as the original code alters its behavior by changing parts of itself. Self-modifying code might seem rare today, but something else can be regarded as self-modification as well: Operating systems load new code into memory, dynamic recompilers generate code at runtime, and some applications decrypt or decode machine code before execution. Neither of these can therefore be statically recompiled.
UQBT, the University of Queensland Binary Translator [24], is a very powerful static recompiler with multiple source and target languages. It is one of the most well-known research projects on binary translation. The ”staticrecompilers” group around Graham Toal and Neil Bradley [26][27] is developing a recompiler to statically translate classic 8 bit arcade games into C code, which is then compiled into native code using a standard C compiler. Finding out all code within the executable is done by playing the game from the beginning to the end, finding all secrets, and logging what code is reached. Using their recompiler, the game ”Ms. Pacman”, which had been designed for a 4 MHz Z80 CPU, runs 450 times faster on a 2.4 GHz i386 than on the original hardware, which is about twice the speed achieved by an optimized interpreter implemented in assembly [57].
Microsoft’s Java competitor, the .NET system, also includes a static recompiler. While .NET executables are typically dynamically recompiled (i.e. at runtime), ”ngen.exe”, which ships with the .NET runtime environment, statically translates MSIL (”Microsoft intermediate language”) code into native code and replaces the original executable with the translated version. Complete translation is always possible, because things that can break static recompilation are impossible in MSIL. The security/safety restrictions of both the high level language (C# etc.) and MSIL prohibit things such as jump tables of unknown size, self-modifying code and code/data ambiguity. Since compilation from high-level source code into MSIL does not discard any data (other than comments and variable names), ngen.exe might be regarded as the back end of a compiler rather than a real recompiler, though.

2.1.4.2 Dynamic Recompilation

In the case of dynamic recompilation (or ”just in time”, JIT translation), the recompilation engine is active while the application is being executed, and can take over execution at any time, if more code has been discovered or code has been generated or modified, to continue recompiling. This solves the problem inherent in static recompilers.
Dynamic Recompilation can also be used effectively even when self-modifying code is present, as long as such modification is infrequent. This is done by marking memory pages on the host that contain the host (translated) code as read-only. When the page is written to, an exception occurs. This exception can be used to mark the page as ”dirty” and needing later recompilation with the new data.

Figure 2.7: Dynamic Recompilation

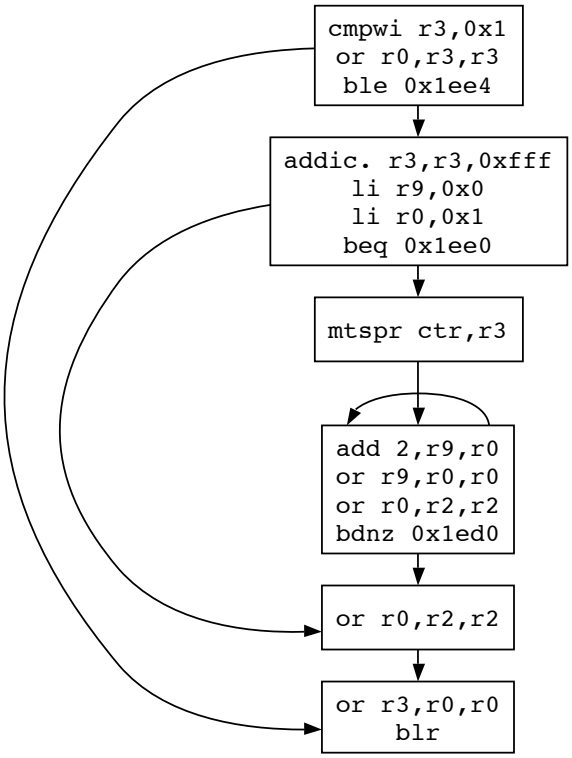


The disadvantage of dynamic recompilers is that they only have limited time to do the translation, as it will be done just before or during execution of the application. The better the optimization that is done, the faster the resulting code will be - but the longer translation takes. As translation and execution time add up to the effective time the user has to wait, extensive optimization will probably not be worth it. A compromise between translation speed and speed of the generated code has to be found to maximize the effective execution speed.
Because of this potential performance problem, there are several techniques to circumvent it. As there is no need to translate the complete program at the beginning, most dynamic recompilers only translate a block of code at a time, execute it, then translate the following block and so on. This makes sure that only blocks are translated that are ever reached. Functionality of an application that is not used by the user will not be translated. These blocks of code are typically ”basic blocks”. A basic block is a sequence of code that is atomic in terms of code flow, i.e. that can be entered only at the beginning, and exited only at the end. So it can only contain up to one instruction of code flow (jump, branch, call...), which must be the last instruction of the block, and there are no instructions in other (known) blocks that jump to any other instruction of this block than the first one[5]. In the following example, the borders between basic blocks have been marked:

```
-------------------------------------
 00001eb0        cmpwi   r3,0x1
 00001eb4        or      r0,r3,r3
 00001eb8        ble     0x1ee4
-------------------------------------
 00001ebc        addic.  r3,r3,0xffff
 00001ec0        li      r9,0x0
 00001ec4        li      r0,0x1
 00001ec8        beq     0x1ee0
-------------------------------------
 00001ecc        mtspr   ctr,r3
-------------------------------------
 00001ed0        add     r2,r9,r0
 00001ed4        or      r9,r0,r0
 00001ed8        or      r0,r2,r2
 00001edc        bdnz    0x1ed0
```

[5]In practice though, many recompilers use bigger basic blocks, for optimization purposes, although it can make recompilation slower and lead to code that has been translated more than once.

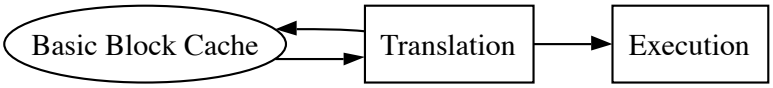
Figure 2.8: A Control Flow Graph




```
-------------------------------------
 00001ee0        or      r0,r2,r2
-------------------------------------
 00001ee4        or      r3,r0,r0
 00001ee8        blr
-------------------------------------
```

Every block in the example above is atomic in terms of code flow; there is no instruction that gets jumped to that is not the first instruction of a block. The following code flow diagram illustrates this:
A dynamic recompiler can translate basic blocks as an atomic unit. The state between instructions in a basic block has no significance. A basic block is the biggest unit with this kind of atomicity, but with some constraints, bigger blocks are possible: For example, instead of cutting blocks at the first control flow instruction, it is possible to cut them at the first control flow instruction that points outside the current block.
There is certainly no need to translate the same block twice, therefore a dynamic recompiler has to cache all blocks that have already been translated. If one basic block has been executed, it has to be looked up whether the next one is already recompiled. If it is, it gets executed, if it is not, it gets translated and then executed.
Dynamic recompilers are a lot more common than static ones. The most prominent example is probably the Java Virtual Machine in many of its implementations (including Sun's). Apple's "DR" (Dynamic Recompiler), which simplified the migration of Motorola M68K

Figure 2.9: Basic Block Cache



to PowerPC processors in Mac OS after 1994, and still ships with the latest version of Mac OS X, can even run system components and drivers correctly. The open source scene has produced a lot of dynamic recompilers for different types of source and target CPUs, many of which have been used in a number of different emulators. Bernie Meyer's Motorola M68K to i386 recompiler of the Amiga Emulator UAE, for example, has also been used in the Macintosh M68K emulator Basilisk II [17] and the Atari ST emulator Hatari [18]. On a 166 MHz Pentium MMX, UAE/JIT achieves about the same performance as a 50 MHz 68060 [58].

2.1.5 Hotspot: Interpretation/Recompilation

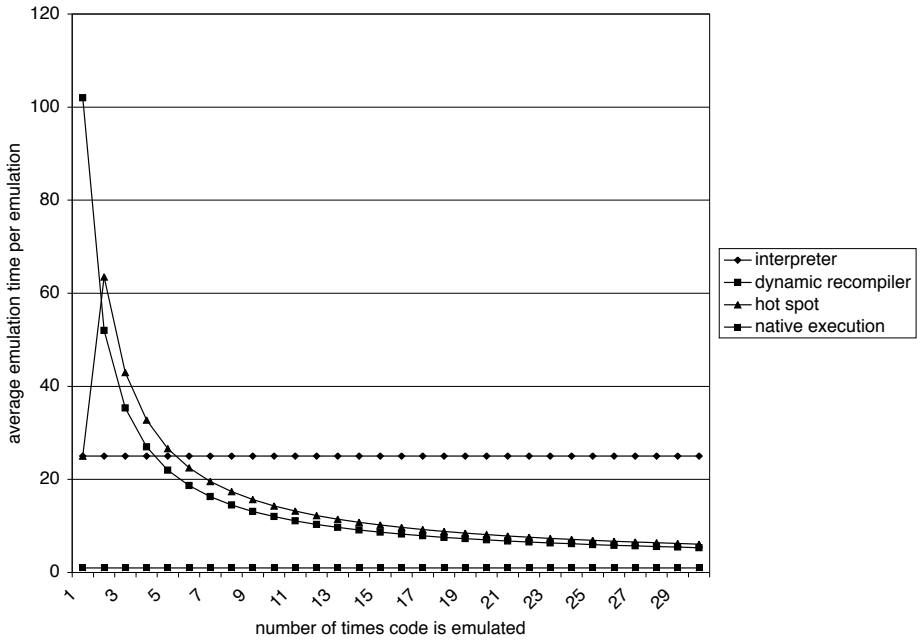
> "We try and avoid the word [emulator]. When people think of emulators they think of things that are very slow." - *Bob Wiederhold, CEO Transitive Corp.*

Recompiled code is faster than interpreting code, and dynamic recompilation is more compatible than static recompilation, so dynamic recompilation seems to be the most sensible choice for emulators. But dynamic recompilation is not necessarily faster than interpretation: It moves work out of loops, but the work that has to be done per instruction is increased: Translating an instruction and executing it clearly takes longer than interpreting it, because the translator also has to generate machine code for the host platform. Therefore recompilers are slower than interpreters if all instructions are only executed once.
The principle of locality states that the CPU spends 90 % of its time with 10 % of the code - and 10 % of its time with 90 % of the code. As a consequence, only 10 % of the code really needs to execute fast, the rest of the code might only be executed a few times or even just once, so recompilation would not amortize. Moving the dispatching and decoding work away from the execution of an instruction should only be done if it is moved outside a loop. It is quite impossible to find out what instructions will be executed more often without actually executing the code, so one easy algorithm is to interpret code, until it has been executed a certain number of times, then recompile it. Code that does not get executed a certain number of times will never be recompiled. This is called the "hotspot" concept.
[20, p. 27] demonstrates this with the following formulas:

Interpreter	nc_i
Dynamic Recompiler	$c_r + nc_e$
Hotspot Method	if $n > t$ then $tc_i + c_r + (n-t)c_e$ else nc_i

c_i	cost of a single interpretation of the code
c_t	cost of recompilation of the code
c_e	cost of running the translated code
n	number of times the code is to be emulated
t	number of times the code is interpreted before it gets translated

Figure 2.10: Speed of Interpreter, Dynamic Recompiler and Hotspot Method



$$c_e = 2,\ c_i = 25,\ c_r = 100,\ t = 1$$

The average cost of one execution is calculated by dividing the result by n. Figure 2.10 illustrates the formulas.
In this case, dynamic recompilation is faster than interpretation if code is run at least five times, but code that is only executed once has incredibly high cost. The hotspot method also amortizes after five runs, but only has the costs of an interpretative run in case the code is only executed once. After 30 runs already, the hotspot method is only 14 % slower than dynamic recompilation, after 1000 iterations, the difference is down to 1 % .
Sun's Java Virtual Machine is an example of a hotspot compiler[10]. It interprets Java bytecode for a while to detect hotspots, and compiles methods later based on the data gathered. The approach of FX!32, a solution that makes it possible to transparently run i386 Windows code on the Alpha version of Windows, is somewhat different: When an application is first executed, it is exclusively interpreted, and runtime behavior information is gathered. A background process can then use the data to translate the application offline, without having any time constraints. This is particularly effective with GUI applications, because they spend a considerable amount of time waiting for user input.

2.1.6 Liveness Analysis and Register Allocation

Liveness analysis and register allocation seems to be for compiler construction what scheduling algorithms are for operating systems: Few people actually implement it, but it is a major topic in the corresponding university lectures. But if a recompiler wants to generate very fast code, it has to do liveness analysis and register allocation.

Table 2.3: Liveness of the variables in the example program

	a	b	c
1			X
2	X		X
3		X	X
4		X	X
5	X		X
6			X

For optimization purposes, modern high-level language compilers try to make optimal use of the CPU's registers, so they try to map as many local variables of a function to registers, in order to minimize the number of memory accesses. As registers are limited, and a function can have more local variables than the machine has registers, a decision needs to be made what variables should be mapped to registers. Also, just selecting a subset of the variables and map every variable to one register is inefficient: A variable might not be in use during the complete function, so the register can be reused for another variable when the original variable is not in use. Speaking more generally, variables whose lives do not overlap can be mapped to a single register.

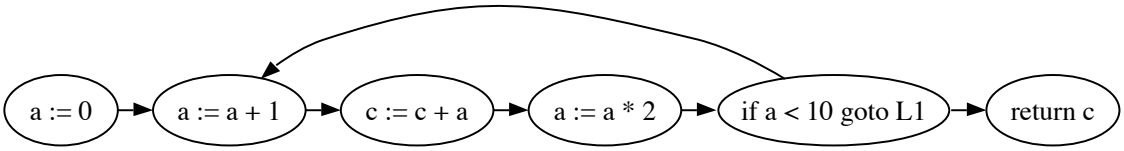
2.1.6.1 Liveness Analysis

Liveness analysis is the technique to find out when a variable is live and when it is not. The following example [46][47] demonstrates this:

```
1     a := 0
2 L1: b := a + 1
3     c := c + b
4     a := b * 2
5     if a < 10 goto L1
6     return c
```

At first sight, it looks like the function needs three variables for the three registers a, b and c. But in this case, two registers are enough, because the lives of a and b do not overlap.
A variable is live at a certain point of the function if there is a possibility that the value it is containing will be used in the future, else it is dead. More formally, it is live if a path (including jumps and conditional branches) exists from this point of the function to an instruction which reads the variable, and the variable does not get assigned a new value along this path. As an assignment overwrites the contents of a variable, the old value is no longer relevant, so it cannot have been live before. Table 2.3 shows the lives of the variables in the example above.
a is live in 2 and 5, because it gets read in these instructions. Instruction 4 overwrites a, so a is dead in instruction 4, but also in instruction 3, because the next access is the write access in 4, and there will be no read accesses earlier than instruction 4. a is also dead in 1, because it gets written there, and in 6, because it doesn't get read any more after instruction 5.

Figure 2.11: The Control Flow Graph of the Example



b gets read in instructions 3 and 4, so it is live there. b is assigned a value in 2, so it is dead at instruction 2 and before, and because b is not read any more after instruction 4, it is dead in 5 and 6 as well.
c is live in instruction 3, because it gets read there. As it is read in instruction 6 without being written between 3 and 6, it is also alive in 4, 5 and 6. The ”goto” in instruction 5 can also branch to instruction 2, which leads to a read access of c in instruction 3 again, so instruction 2 is on a path to a read access without write access in between, and c is alive in instruction 2 as well. In this case, c is obviously a parameter of the function, because it gets read in instruction 3 before if it written for the first time. Therefore c must be alive in 1 as well.
The table shows that a and b do not overlap, so they can be assigned a single register. The following source code would even be equivalent to the original one:

```
1     a := 0
2 L1: a := a + 1
3     c := c + a
4     a := a * 2
5     if a < 10 goto L1
6     return c
```

Now all this has to be translated into an algorithm. A control flow graph (CFG) helps describe a function for this purpose. A CFG is a directed graph whose nodes are the individual statements of the function. Edges represent potential flow of control. The CFG of the example is shown in figure 2.11.
The two arrays succ[] and pred[] contain the sets of potential successors and predecessors of each node. For example, succ[3] = { 4 } , succ[5] = { 2,6 } , pred[6] = { 5 } and pred[2] = { 1,5 } . For the formal definition of liveness two more arrays are needed: def[] contains the set of variables written (”defined”) by a specific instruction, and use[] contains the set of variables read (”used”) by a specific instruction. For instance, use[3] = { b,c } and def[3] = { c } in this example. Table 2.4 is the complete succ/pred/use/def table.
The formal definition of liveness is as follows: A variable v is live at a certain node n if a path exists from n to node m so that $v \in \text{use}[m]$ and foreach $n <= k < m$: $v \notin \text{def}[k]$. The liveness analysis algorithm outputs the arrays of sets in[] (variable is live before instruction) and out[] (variable is live after instruction). The following rules help design the algorithm:

1. $v \in \text{use}[n] \Rightarrow v \in \text{in}[n]$

   If a variable is used by an instruction, it is alive just before the instruction.

Table 2.4: succ[], pred[], use[] and def[] for the example program

	succ	pred	use	def
1	2			a
2	3	1,5	a	b
3	4	2	b,c	c
4	5	3	b	a
5	2,6	4	a	
6		5	c	

2. $v \in (\text{out}[n] - \text{def}[n] \Rightarrow v \in \text{in}[n]$

   If a variable is live just after an instruction, and it is not written to by the instruction, it is live just before the instruction.

3. $\text{foreach}\, s \in \text{succ}[n] : v \in \text{in}[s] \Rightarrow v \in \text{out}[n]$

   If a variable is live just before at least one of the successors of the current instructions, it is live just after the current instruction.

The liveness analysis algorithm looks like this:

$\text{foreach}\, n : \{$
    $\text{in}[n] = 0$
    $\text{out}[n] = 0$
$\}$
repeat $\{$
    $\text{foreach}\, n$: $\{$
        $\text{in}'[n] = \text{in}[n]$
        $\text{out}'[n] = \text{out}[n]$
        $\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \setminus \text{def}[n])$ // (rules 1 and 2)
        $\text{out}[n] = \bigcup_{s \in succ[n]} in[s]$ // (rule 3)
    $\}$
$\}$ $\text{until}\, \text{in}' = \text{in} \wedge \text{out}' = \text{out}$

The algorithm repeatedly looks for new elements for in[] and out[] and terminates if there have been no new additions. Within one iteration, the three rules of register allocation are applied to all nodes. The result of the example is summarized in table 2.5.
A variable v is live at instruction n if $v \in \text{in}[n]$. The complexity of this algorithm is between $O(n)$ and $O(n^2)$, n being the number of instructions of the function [47].

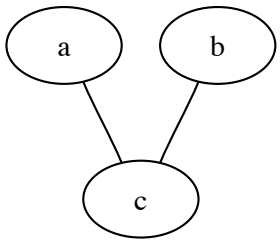
2.1.6.2 Register Allocation

Now that we have the liveness of each variable, it is easy to find out which variables interfere: The interference graph consists of nodes, which represent the variables, and edges, which represent interference, i.e. the connected variables exist at the same time and cannot be mapped to the same register. The graph is constructed using the following algorithm:

Table 2.5: Results of the liveness analysis algorithm

	succ	pred	use	def	in	out
1	2			a	c	a,c
2	3	1,5	a	b	a,c	b,c
3	4	2	b,c	c	b,c	b,c
4	5	3	b	a	b,c	a,c
5	2,6	4	a		a,c	a,c
6		5	c		c	

Figure 2.12: The Interference Graph of the Example




```
foreach n: {
    foreach a = def[n]: {
        foreach b = out[n]: {
            connect(a, b)
        }
    }
}
```

For every instruction n, every variable that is written to ($a \in \text{def}[n]$) is connected to every variable that is alive just after the instruction ($b \in \text{out}[n]$). One exception optimizes the interference graph: If an instruction is a move command "a = b", there is no need to add the edge (a,b). The interference graph of the example is shown in figure 2.12.
Assigning n registers to the nodes of the interference graph now means coloring the graph with n colors so that no adjacent nodes have the same color. The algorithm to achieve this has two phases. In the simplification phase, all nodes are removed one by one and pushed onto a stack. In every iteration, any node with less than n neighbors will be selected and removed. If the graph has no node with less than n neighbors, any other node will be selected and removed. This node is a spill candidate, i.e. it is possible that this variable cannot be assigned to a register and must reside in memory, so a variable that is infrequently used should be selected.
In the selection phase, the stack is completely reduced one by one, rebuilding the graph. Every node that is reinserted into the graph will be assigned a color, i.e. a register. If a node cannot be colored, it will not be inserted and spilled to memory instead. The complexity of the register allocation algorithm alone is between $O(n)$ and $O(n^2)$, n being the number of variables. In practice, the speed of all algorithms together very much depends on the speed of liveness analysis, because the number of instructions is typically a lot higher than the

number of variables.

2.2 Endianness

> ”It is computed that eleven Thousand Persons have, at several Times, suffered Death, rather than submit to break their Eggs at the smaller End. Many hundred large Volumes have been published upon this Controversy.” - *Jonathan Swift, Gulliver’s Travels*

Endianness (”byte order”) describes the order of bytes (or bits) if multiple bytes (or bits) are to be stored, loaded or transferred. There is no ”natural” endianness, instead, it is a matter of convention.
A CPU whose bus transfer is limited to (aligned) machine words has no endianness[6]. Otherwise, there need to be conventions:

- If the CPU wants to access a value that has twice the width of a machine word, will the low or the high word be stored at the lower address in memory?
- If the CPU wants to access parts of a machine word (for example, a byte) in memory, does reading from the address of the word read the lowermost or the uppermost bits?

The little endian convention stores the least significant data first. When dividing a 32 bit value into 8 bit fields, the 32 bit value 0x12345678 would be stored as 0x78, 0x56, 0x34, 0x12. Although most desktop computers in use today use the little endian system, it is only because most desktop computers are i386 systems. The DEC PDP-11 (the original UNIX machine), the DEC VAX (VMS), the MOS 6502+ (8/16 Bit Commodore, Apple, Atari, Nintendo) and all Intel main branch CPUs like the 8080 (CP/M), 8086 and the i386 line (i386, i486, Pentium, ...) are little endian designs.
The advantage of the little endian byte order is that operations like additions on huge data can be split into smaller operations by handling byte for byte starting with the address of the value in memory up to higher addresses. This example in 8 bit 6502 code adds two 32 bit values in memory:

```
clc
lda 10
adc 20
sta 30
lda 11
adc 21
sta 31
lda 12
adc 22
sta 32
```

[6] If values that exceed the width of a machine word has to be processed, the application defines the endianness.

```
lda 13
adc 23
sta 33
```

This is especially useful when working with very huge data in a loop. The 32 bit addition on the 8 bit CPU in the example could also have been done in a loop, counting up.
Furthermore, with little endian encoding, casting between different sizes is faster: In order to cast a 32 bit value in memory to 16 bit, it is enough to read a 16 bit value from the same address, as the least significant bit of the value is stored at a fixed position.
Big endian CPUs store the most significant data first. When dividing a 32 bit value into 8 bit fields, the 32 bit value 0x12345678 would be stored as 0x12, 0x34, 0x56, 0x78. Many CPUs of the past and the present are big endian, such as the Motorola 6800 and 68000 (M68K) series (32 bit Amiga, Atari, Apple, Sega, Sun), the HP PA-RISC (UNIX) and the SPARC (Sun). Although most modern CPUs can be switched between both conventions, they are typically operated in big endian mode (MIPS, PowerPC)[7].
The main advantage of the big endian encoding is that memory dumps are easier to read for humans, data can be converted into ASCII text more easily and the sign of a value can be found out more easily, as the most significant bit is stored at a fixed position.
There is no noticeable performance difference between both conventions. Little endian might seem a little more natural in a mathematical sense, but practically it is only a matter of taste. It does not matter whether a machine is big or little endian - but endianness can be a severe problem when machines of different byte order have to interact with each other, especially when data is transferred from one machine to another machine that has a different endianness, or when a program written for one machine is supposed to be run on a machine with a different endianness. In the first case, all 32 bit values would have the wrong byte order and the bytes would have to be swapped in order to retrieve the correct value. But ASCII strings for example would be unaffected, as they are read and written byte by byte, so they must not be converted. Unfortunately there is no way to tell what to convert and what not to convert when changing the endianness of data.
In case the target CPU does not support the byte order of the source CPU, an emulator has to address the endianness problem manually. This is typically done by byte swapping all data that is read or written.

2.3 Intel i386

> ”The x86 isn’t all that complex - it just doesn’t make a lot of sense.” - *Mike Johnson, Leader of 80x86 Design at AMD, Microprocessor Report (1994)*

> ”8086 is the hunchback among processors” - *Halvar Flake [50]*

The Intel i386 line is a family of 32 bit CISC CPUs that has been first released in 1985, as a backwards compatible successor to the popular Intel 8086 line. There is no consistent

[7] Probably because UNIX systems are typically big endian, and because some network technologies like Ethernet and TCP/IP store data in big endian byte order.

name for this architecture. Intel, who used the name ”Pentium” for almost all CPUs from this line since 1992, started referring to it as IA-32 (Intel Architecture, 32 bit) since the 1990s, AMD [68] and VIA [69], who are producing compatibles, refer to it as the ”x86 line”. Elsewhere it is also called the i386 or the 80386 line.
The term ”i386” has been chosen to be used in the following chapters, because IA-32 and Pentium are both Intel marketing terms, and because ”x86” also includes the more primitive 16 bit versions - this work only deals with the 32 bit instruction set of the i386 line.

2.3.1 CISC

The i386 is a very typical example of a CISC processor. The term CISC means ”Complex Instruction Set Computing” and has actually been first used in the 1980s to differentiate the new RISC design from the traditional 1970s design. These CPUs have two major characteristics:

1. **Complex instructions**: As compiler technology had not been advanced enough, most projects had been implemented in assembly language, as it achieved significantly better performance. But assembly language is more complicated to develop in, among other things because the increased number of lines of code makes it harder to keep track of the algorithms. Therefore CPUs were designed that made it easier for the developer, by providing more powerful instructions and addressing modes that made code shorter and more readable. This was accomplished with the system of microcoding: The CPU consists of a microcode interpreter and a microcode ROM, which in turn interprets the complex instruction set. This design also made it easier for the relatively primitive compilers [8] of that time to generate efficient code. Thus complexity has been shifted from software to hardware.

2. **Few registers**: What makes computers expensive today are high-performance CPUs - in the 1970s, it was memory. While today, computers can waste virtually as much memory as they want to, memory was very limited back then, so everything had to be done to make maximum use of it. In a world of 4 KB of RAM, this also meant that machine code had to be very compressed[9]. As a consequence, encoding was quite complicated, with some special encodings occupied with very different instructions, to save bits, and the number of registers was very limited. 16 registers, for example, would have meant that 4 bits were needed to encode a register number - if all instructions with only one register as an operand were supposed to occupy only one byte, then there would only be 4 bits left to encode the actual operation. As a compromise, many CPUs had more registers, but allowed certain operations only on some registers. For example, indexed addressing was only possible with four

[8] Compiler technology had already been quite advanced by the 1970s, but complex compilers were not yet available for microprocessors, which were the low end computing line at that time, as opposed to IBM System/360-style mainframes.

[9] In 1977 for instance, Microsoft ported their famous 8K BASIC interpreter for the Intel 8080 onto the MOS 6502, which has a less compact instruction coding. The 6502 version did not fit into 8 KB, but occupied 9 KB, which was disastrous at that time, because computers were typically equipped with 8 KB of ROM.

(BX/BP/SI/DI) of eight registers on the 8086, and the Motorola M68K had two sets of eight registers, one set for arithmetic, one for memory accesses and indexing.

2.3.2 History

The 8086 line of CPUs evolved over more than 25 years, and has its roots even further in the past. There was no single design group, no formal definition of a new direction. Instead every extension was based on what was technologically possible at that time, always remembering not to break backwards compatibility or going too far into new directions. The following summarized history of the 8086 line of CPUs is supposed to illustrate the chaotic design of these CPUs [30] [31]:

- 1974: The 8080, which is similar to the 8008 (whose design has been specified by Intel's customer Computer Terminal Corporation) from two years earlier, is released. It has a 16 bit address bus (64 KB) and eight 8 bit registers A, F (flags), B, C, D, E, H, L, of which the B/C, D/E and H/L combinations could be used as 16 bit registers. The 8080 and its immediate successors and clones (8085, Zilog Z80) are very popular CPUs and are a requirement for the most successful 8 bit operating system, Digital Research CP/M.

- 1978: The 8086 is released. It has a 20 bit address bus (1 MB) and eight 16 bit (non-general-purpose) registers AX, BX, CX, DX ("X" means "extended"), SI, DI, BP, SP. The former four registers can be split into eight 8 bit registers (AL, AH, BL, BH, CL, CH, DL, DH). It is not binary compatible with the 8080, but it is compatible on the assembly level: By mapping A to AL (lower half of AX), HL to BX, BC to CX and DE to DX, 8080 assembly code could be transformed into 8086 assembly code easily by an automated program [59]. Also the memory model is compatible, as the 8086 addresses its memory in 64 KB "segments". The 8086 was supposed to simplify the migration from 8 bit CP/M to 16 bit CP/M-86. The 8086 has its breakthrough in 1982 with the the IBM PC, running with MS-DOS, a CP/M-86 clone by Microsoft.

- 1980: The 8087 floating point extension unit is released, which has been designed by an independent team in Israel instead of the 8086 team in California. As the 8087 is only an extension to the 8086, instructions are fetched by the 8086 and passed to the 8087. Since the protocol between the two chips requires the data to be minimal, instructions for the 8087 cannot be long enough to contain two registers as operands, so the designers chose to create a floating point instruction set that needs one operand at most, by equipping the 8087 with a stack consisting of 8 registers. Binary instructions can either work on the two uppermost values on the stack, or with the value on the top-of-stack, and another value taken from any position in the stack. Stack overflows are supposed to be handled in software, using exceptions, to simulate a stack of an arbitrary size. Unfortunately, a design flaw [10] made this idea, which certainly doesn't excel in speed, ultimately uneconomic.

[10] Instead of returning an address pointing directly to the faulty instruction, so it could be executed again, the exception returns an address that points after it.

- 1982: The 80286 is released, which sports a 24 bit address bus (16 MB), a more sophisticated memory model and simple memory protection. The CPU can switch between ”real mode” (8086 compatible) and ”protected mode” (new memory model and protection). Two design flaws in the 80286 are patched by IBM in its IBM AT computers: As the address bus is now 24 bits wide, real-mode addresses above 1 MB do not wrap around to the bottom of RAM, as some 8086 applications required. This is fixed by IBM’s A20 gate, which can be instructed to hold down the CPU’s address line 20 to zero, simulating the effect. Intel moved this feature into the CPU later and is still present in today’s i386 CPUs. The other flaw made it impossible to switch from protected mode into real mode[11]. Many old (real mode) applications wanted to access memory above 1 MB by switching into protected mode, copying memory and switching back, but this was only made possible when IBM added functionality that would set a flag in the NVRAM, soft-reset the CPU, detect in the startup ROM whether the flag had been set, and continue execution in the original code if it was the case. This is fixed in the i386 with a direct way to exit protected mode, though the reset method is still supported by ROM for compatibility.

- 1985: The i386 is released. It has eight 32 bit registers, EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP (”E” means ”extended”, once again, so ”EAX” is ”extended” accumulator ”extended”), which are a lot more general purpose than the 8086’s registers, improved and orthogonalized instructions and addressing modes, a (theoretical) 32 bit address bus (4 GB) and virtual (paged) memory. The i386 can still access the 16 (”AX”) and 8 bit parts (”AL, ”AH”) of all registers and still has the 8086 real mode and the 80286 protected mode next to the new 32 bit protected mode and the virtual 8086 mode. However, a design flaw of the 32 bit protected mode made the i386 unsuited for many modern operating systems like Windows NT, because when executing in kernel mode, write-protection on memory pages is not enforced[12]. Other than switching temporarily to 32 bit protected mode to copy between the low 1 MB and the extra memory above 1 MB, and some games running on so-called ”DOS Extenders”, the i386 extensions were hardly used for 10 years, until the release of Windows 95. Yet, the i386 extensions have been the most important so far to the 8086 line.

- Between 1985 and 1997, although the i386 line of CPUs was drastically optimized internally with each new CPU, there were no significant extensions to the i386 on the ISA level. The i486 (1989) and the Pentium (1992) introduced only a few new instructions (dealing with semaphores[13] and endianness), and the Pentium Pro (1995)

[11] Actually, it was later discovered that a secret Intel instruction, known as ”loadall”, was capable of exiting protected mode into real mode. However, by the time this knowledge reached the public, the i386 had already been released.

[12] Newer operating systems use the ”copy-on-write” memory strategy frequently, by marking pages as read-only to detect writing, but with this misguided i386 ”feature”, the operating system had to manually check write permission in system calls, a huge bottleneck. The i486 corrected this by adding a new mode, disabled by default for compatibility with the i386, that causes write-protection to be fully enforced even in kernel mode.

[13] The Pentium’s ”cmpxchg8b” brought along with it a security bug (”F00F”) that nearly forced Intel to

added a conditional move instruction.

- Since 1997, a number of additional SIMD units have been added in order to accelerate games and video decoding: 1997 MMX (integer SIMD)[14], 1998 3Dnow! (floating point SIMD, introduced by AMD), 1999 SSE (floating point SIMD, competitor to 3Dnow!, but incompatible), 2001 SSE2 (64 bit floating point SIMD, 128 bit integer SIMD) and 2004 SSE3 (additions to SSE2). All the additional units are specialized on multimedia processing and only include commands that work on special registers, so compilers rarely use the new units. The i386 integer instruction set was unchanged in this period.

- 2003: The AMD Opteron is released. While Intel has decided in the 1990s already to abandon the i386 architecture for the VLIW (”very long instruction word”) architecture IA-64 (”Itanium”) as soon as the step to 64 bit is due, AMD decided to step in and design a 64 bit backwards-compatible i386 successor whose 64 bit mode was similar enough to make ports of operating systems, compilers and applications easy. Just like the i386 extended the architecture to 32 bit, the ”AMD64” (formerly ”x86-64”) extensions extend the i386 architecture to 64 bit and add another eight registers. The register names are now RAX [15], RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14 and R15. AMD64 still supports access to the lower 32 bit of a register (”EAX”, this is possible for the first eight registers only), to the lower 16 bit (”AX”, only possible with the first eight registers) and accessing the lower two bytes (”AL”, ”AH”, only possible for the first four registers), all old modes (real mode, 80286 protected mode, 32 bit protected mode, virtual 8086 mode) and the new 64 bit protected mode, and uses the same instruction encoding as the i386, extended with prefixes to differentiate between 32/64 bits and between the lower and upper eight registers. Intel copied the technology under the name EM64T and releases a 64-bit enhanced Pentium 4 in 2004 - but its implementation is based on an earlier AMD specification, so it is not fully compatible. Linux, OpenBSD, FreeBSD and NetBSD have been available since the day of the introduction of the first AMD64 CPU; a release version of Windows for AMD64/EM64T is planned in 2005.

Let’s summarize: The latest 64 bit CPUs are extensions of a 32 bit architecture (by Intel) based on a 16 bit architecture (by Intel) that had been designed to be somewhat compatible with an 8 bit architecture (initially designed by Computer Terminal Corporation). The floating point design was based on properties of the 8086 interface and slowed down applications until the advent of SSE2 in in 2001, which superseded the old stack. The latest i386 CPUs use the 8086 instruction encoding, enhanced with prefixes to differentiate between 16/32/64 bit memory and data, and to access the upper eight registers. Even the Athlon 64 and the Pentium 4/EM64T still start up in 16 bit real mode - also the PC-based gaming console Microsoft Xbox executes its first few cycles in 16 bit code. Nevertheless, the i386

recall the Pentium a *second* time.

[14]The new eight 64 bit registers are, though bad for performance, physically mapped to the 8 floating point registers, so that operating systems do not have to be patched to save more registers on a context switch

[15]neither ”EEAX” nor ”EAXX”

Table 2.6: Examples of differences in Intel and AT&T syntax

mov eax, ebx	movl %ebx, %eax
mov eax, 17h	movl $0x17, %eax
mov eax, dword ptr 23	movl 23, %eax
mov [eax+ebx*8+23], ecx	movl %ecx, 23(%eax,%ebx,8)
jmp ebp	jmp *%ebp

is the most successful architecture on the desktop, mainly because of its DOS/Windows success.
The 8008/8080/8086 heritage is important in order to understand some of the quirks of the i386, but in the following chapters, only the 32 bit i386 architecture without the floating point or SIMD units will be of importance as its instruction set is the common denominator of the i386 architecture and it is used by virtually all currently available applications designed for the 8086 line.

2.3.3 AT&T and Intel Syntax

There are two different conventions for the i386 assembly language: the Intel syntax and the AT&T syntax. The latter was named after the creator of UNIX, as the AT&T syntax obeys the syntax rules of the standard UNIX assembler ”as”. Please note that all this is only about syntax - corresponding instructions always produce the same machine code. This chapter is supposed to present the differences between the two notations.
The most important difference is the order of the operands: The Intel convention has the target as the first operand (a += b like notation), and the AT&T convention has this reversed (move from-to notation). Indirect addressing is indicated by ”[]” in Intel and by ”()” in AT&T notation. Constants are prefixed with a dollar sign in AT&T syntax only; Intel demands for ”byte/word/dword ptr” instead in case it is an address instead of an immediate. In addition, AT&T requires the ”%” prefix for every register and a ”b”/”w”/”l” suffix to the mnemonic if the data size would else be ambiguous. Also AT&T uses the C notation (”0xff”) for hexadecimal values as well as a different syntax for complex indexed addressing modes and requires a ”*” for indirect jumps. The examples in table 2.6 illustrate the differences.
The AT&T syntax is more logical, less ambiguous, and is the standard i386 syntax on UNIX systems. The following chapters will therefore make use of the AT&T syntax exclusively.

2.3.4 Registers

The i386 has eight 32 bit registers which are more or less general purpose: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI[16]. Table 2.7 summarizes the original purposes of the specific registers on the 8086.

[16]The order is not alphabetical, as this is their internal numbering: The registers were most probably not designed to be A, B, C and D; instead, four registers have been designed and names have been assigned to them later. Clever naming lead to A, B, C and D, which did not conform to the original order.

Table 2.7: i386 registers and their original purposes

(E)AX	accumulator (”mul”/”div” always imply this register)
(E)BX	base register (for indexed memory accesses)
(E)CX	count register (the ”loop” instruction works with (E)CX only)
(E)DX	data register (”mul”/”div” always imply this register)
(E)SP	stack pointer
(E)BP	base pointer (points to beginning of stack frame, instructions ”enter”/”leave”)
(E)SI	source index (source pointer for string operations like ”movs”)
(E)DI	destination index (destination pointer for string operations like ”movs”)

On the i386, these registers are pretty much general-purpose. Some deprecated instructions like ”enter”/”leave” and ”loop” use specific registers, but all instructions that are still typically used today work with any of the registers. There are two exceptions to this: ”mul”/”div” still always imply EAX/EDX, and some complex ”scale indexed” addressing modes do not work with the stack pointer ESP. Although ESP can be used in any other instruction or addressing mode (it can contain the source value for a multiplication or be shifted right, for example, which is not typically useful for a stack pointer), it is not practical to use it for any other purpose. In some modes of the CPU, exceptions and interrupts for example will always push the flags onto the stack using ESP as a stack pointer, and some operating systems also assume ESP to point to a stack.

The lower 16 bits of each of the eight 32 bit registers can be accessed by omitting the ”E” from the register name. For EAX, EBX, ECX and EDX, it is also possible to access the upper and the lower 8 bits of the lower 16 bits separately: AH, AL, BH, BL, CH, CL, DH, DL. (There is no way to access the upper 16 bits of a registers directly or to split the upper 16 bits as it is possible with the lower 16 bits.) So the i386 can effectively natively work with 8, 16 and 32 bit values.

There are two more special purpose user mode registers: The 32 bit program counter EIP (instruction pointer), which cannot be addressed in any other way than by explicit control flow instructions, and the 32 bit EFLAGS registers, of which only four bits (sign flag, overflow flag, zero flag, carry flag) are typically used in user mode[17]. EFLAGS can only be accessed by placing it onto or taking it from the stack, or by copying the lower 8 bits to or from the AH register.

2.3.5 Addressing Modes

It is a typical characteristic of CISC that instructions and addressing modes are pretty much independent, that is, all instructions can be combined with all addressing modes that make sense. This is reflected by the instruction encoding: The opcode is followed by the independent encoding of the addressing mode. Furthermore, CISC often has very complex and flexible addressing modes.

[17] EFLAGS also stores the current mode of the CPU, which makes virtualisation of an i386 pretty tricky, as reading the flags is an unprivileged instruction

Table 2.8: i386 r/m addressing modes

Format	Description	Example	Behavior
b	absolute	mov 0x17, %eax	eax := memory(0x17)
(register)	reg. ind.	mov (%ebx), %eax	eax := memory(ebx)
b(register)	reg. ind. w/ base	mov 0x17(%ebx), %eax	eax := memory(0x17+ebx)
(,register,a)	reg. ind. scaled	mov (,%ebx,4), %eax	eax := memory(ebx*4)
b(,register,a)	reg. ind. scaled w/ base	mov 0x17(,%ebx,4), %eax	eax := memory(0x17 +ebx*4)
(register1,register2,a)	reg. ind., reg. ind. scaled	mov (%ecx,%ebx*4), %eax	eax := memory(eax+ebx*4)
b(register1,register2,a)	reg. ind., reg. ind. scaled w/ base	mov 0x17(%ecx,%ebx,4), %eax	eax := memory(0x17+eax+ebx*4)

2.3.5.1 Characteristics

Depending on the type of the instruction, it can have zero, one, two or three operands, but ”imul” is the only instruction that supports three operands. This means that there can be no operation with two sources and one destination (like a = b + c), instead, one operand is both one source and the destination (like a += b). The same applies to unary operations: Shifts, for example, work on a single register, overwriting the old value.
Instructions with one operand can have, depending on the instruction, an immediate value, or a register or a memory address (”r/m”) as an operand. Instructions with two operands can have register/register, immediate/register, immediate/memory, register/memory or memory/register as operands. This is all combinations except for memory/memory. All these addressing modes are encoded using the ”r/m” encoding.

2.3.5.2 r/m Addressing

The ”r/m” encoding can encode one register as well as another operand, which can be a register or a (complex) indexed addressing mode. Depending on the width of the operation, the width of an r/m operand can also be 8, 16 or 32 bits. A complex indexed addressing mode can have these components:

```
b(register1,register2,a)
```

register1 and register2 are any 32 bit registers (register2 cannot be ESP), the scaling factor a is either 1, 2, 4 or 8, and the displacement b can be 8 or 32 bits wide. This addresses the memory location calculated by $register1 + register2 * a + b$. All components of this format are optional (of course at least register1, register2 or b must be present), so seven addressing modes are possible, as shown in table 2.8.
Some of these addressing modes are very powerful. The following example is supposed to illustrate this: A program stores its data at a (flexible) memory base address, which is stored in register EBX. A data structure of this program is an array of 32 bit integer variables. The offset of this array from the beginning of the flexible memory region has been hard-coded by the compiler, it is 0x10000. The program now wants to increment the value in the array whose index is stored in ECX. The following assembly instruction does exactly this:

```
incl 0x10000(%ebx,%ecx,4)
```

This instruction has to do a shift and an addition of three values to calculate the address, but nevertheless all modern i386 CPUs calculate the address in a single step, that is, the instruction is just as fast as incl 0x10000 or incl (%ebx).

2.3.6 Instruction Set

The instruction set of the i386 can be categorized by the type and number of arguments of the instructions. The biggest category is formed by arithmetic and logic instructions that have two operands. These are mov, xchg, add, sub, adc, sbb, and, or, xor, cmp, test and lea. As mentioned earlier, there is no way to specify three operands, instead, one operand has to be both one source and the target of the operation. There is an exception to this though: The instruction lea (”load effective address”), which has been designed to calculate the address specified using a complex addressing mode, can be abused to do three operand addition, and several more complex operations. The ”lea” instruction is like ”mov”, except that instead of copying the value at the specified memory address to the target register, it copies the address itself. The following instruction is an example of three operand addition abusing ”lea”:

```
lea (%ebx, %ecx), %eax
```

This loads the ”address” EBX+ECX to EAX - so this effectively adds EBX and ECX and stores the result in EAX. This is another form of lea abuse:

```
lea (%ebx,%ebx,4), %eax
```

EBX+EBX*4 is EBX*5, so this multiplication by 5 is of virtually no cost, as the complex multiplication unit is not used.
Some more arithmetic and logic operations only have one operand: dec, inc, neg and not are unary operations. In case the operand is a location in memory, the value gets fetched, modified and written back. The same is true of the shift operations rcl, rcr, rol, ror, sar, shl and shr, but they also have an optional shift count as a parameter.
Multiplication and division are anything but orthogonal: Unsigned multiplication and signed and unsigned division have just a single parameter and implicitly use the registers EAX and EDX as parameters, but signed multiplication takes up to three operands.
The i386 has one characteristic that is quite unusual for today’s CPUs: After every arithmetic or logic operation, the flags (EFLAGS register) get updated with the result of the operation. There is no way to make the i386 leave the flags unchanged.
There is a designated stack addressed by the stack pointer, which is decremented by four by a push, and which always points to the value last pushed. It is possible to push and pop 16 and 32 bit values.
The control flow instructions jmp and call have their operand encoded in ”r/m” format, which makes very complicated calculated jumps possible in a single instruction. 16 different conditional jump instructions (jz, jnz, jc etc.) are available for the most important combinations of flag bits. Their operand is a relative 8 or 32 bit constant. Calling (call) and returning from (ret) functions is done by saving the program counter EIP on the stack and restoring it, respectively.
There are a lot more instructions, most of them have a very irregular encoding and thus very different types and numbers of operands. These include instructions like rep (prefix that repeats the following instruction CX or ECX times), movs (memory copy), cwd (sign extension of AX to EAX), setcc (set byte to 0 or 1 depending on condition).

2.3.7 Instruction Encoding

The i386 instruction encoding is very complex. Instructions can be 1 to 17 bytes in size and consist of an opcode, optional prefixes and, depending on the opcode, optional operands. Prefixes can override the data width of the instruction, cause the following instruction to be repeated ECX times or lock the bus in an SMP environment, for example. Instructions that are available for 8 and 16/32 bit data have a different opcode for 8 than for 16/32 bit. Whether the width is supposed to be 16 or 32 bit is indicated by a one byte prefix in the former case.
Most instructions support ”r/m” addressing modes, so the opcode is followed by an ”r/m” field. Many instructions have special (shorter) encodings if one operand is AL, AX or EAX or an immediate fits into eight bits, in which case it will be encoded as an 8 bit value and sign extended to 16 or 32 bits at runtime.

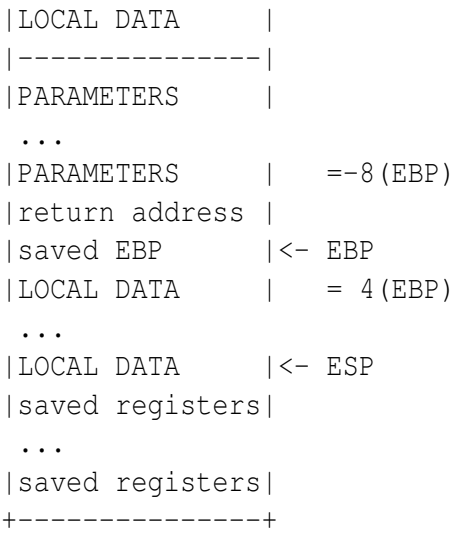
2.3.8 Endianness

The i386 is one of quite few architectures that stores values that occupy more than 8 bit in little endian order and have no option to operate in big endian mode.

2.3.9 Stack Frames and Calling Conventions

There are no definitive calling conventions on the i386, they depend on the operating system (Windows is different than Unix) and sometimes also on the compiler used (Borland and Microsoft compilers on Windows can use different calling conventions between private functions in an application). The following description is valid for all Unix-like operating systems on the i386, like Linux, FreeBSD and Darwin. Inside a function, the stack looks like figure 2.13 (bigger addresses on top).
Because there are so few registers, all parameters are passed on the stack. The caller pushes the parameters onto the stack before calling, in reverse order (addresses 7 to 8). After the called function has returned, the caller has to clean up the stack by removing the parameters it pushed. The called function could access the parameters relative to the stack pointer: ”4(%esp)” would point to the parameter pushed last. But as registers are so few, the called function might have to use pushes and pops to temporarily save a register, so the stack pointer might not be constant within a function. Therefore the stack pointer (ESP) is copied into the base pointer register (EBP) at the beginning of the function, so that all accesses to parameters (and local variables) can be done using EBP-indexing, with positive displacements, as EBP is constant within a function. Of course, the old value of EBP has to be saved on the stack before (5). Local variables are also stored on the stack: The size of all local variables is subtracted from the stack pointer to make room on the stack (3 to 4) that can be accessed EBP-indexed with negative displacements.
Return values are passed in the EAX register for data up to 32 bits and in EAX and EDX for data up to 64 bits. In case of more data, the caller passes an implicit pointer to the location where the called function is supposed to store the data. The EBX, ESI, EDI (and EBP and ESP) registers are volatile and must be preserved, the other registers may be overwritten.
In practice, code to call a function looks like this:

Figure 2.13: i386 Stack Frame




```
push $2
push $1
call function
add $8, %esp
```

This sequence pushes two parameters, ”1” and ”2” onto the stack in reverse order, calls the function and removes the parameters from the stack afterwards by adding 8 to the stack pointer. The function would look like this:

```
push %ebp
mov %esp, %ebp
sub $0x10, %esp
push %ebx
push %esi
...
mov 8(%ebp), %eax
mov 12(%ebp), %ebx
mov %eax, -4(%ebp)
mov %ebx, -8(%ebp)
...
pop %esi
pop %ebx
add $0x10, %esp
pop %ebp
ret
```

First EBP gets saved and the ESP gets copied into EBP. 0x10 is subtracted from the stack pointer to make room for local variables on the stack. As the function might change

EBX and ESI, it pushes these registers. The body of the function can now access the first operand at ”8(%ebp)” and the second one at ”12(%ebp)”. Space for local variables is from ”-4(%ebp)” down to ”-10(%ebp)”. At the end, the function restores EBX and ESI and removes the stack frame by adjusting the stack pointer and restoring the base pointer.

2.4 PowerPC

”The PowerPC is a hybrid RISC/CISC” - *Cristina Cifuentes* [43]

The PowerPC architecture is a formal definition for 32 and 64 bit CPUs, released in 1991 by Apple, IBM and Motorola.

2.4.1 RISC

The PowerPC architecture is an example of a RISC (”Reduced Instruction Set Computing”) CPU. RISC is a concept developed at IBM, Stanford and UC Berkeley in the late 1970s and early 1980s to overcome the typical deficiencies of CPUs of the 1970s [60] [52].
RISC has two main characteristics:

1. Fewer and simpler instructions: Compilers usually generate code that only uses a small part of the instruction set of a CISC processor. Bart Trzynadlowski’s statistics [61] show that the most frequent five instructions[18] are more than 70% of all instructions executed on a Motorola M68K. The idea of the ”quantitative approach to computer design” [31] is to make the common case fast. Complex instructions are removed from the instruction set, which makes the microcode interpreter obsolete and makes decoding easier, even more so as all instructions have the same width. All remaining instructions can be executed in one clock cycle, which makes pipelining and superscalar execution easy. The design is a lot simpler, work has been moved from hardware to software.

2. More registers, load/store architecture: The lack of a microcode interpreter and a complex instruction decoder frees up transistors that can be used otherwise: better pipelining logic, multiple ALUs and, most importantly, many registers. If the CPU has more registers, more operations can be done on registers, so less memory accesses are needed, which can be a significant performance improvement. The term ”load/store architecture” refers to the fact that memory accesses are only possible through load and store instructions; all other operations only work on registers.

The time required for a program to execute is calculated using this formula:

$$\frac{time}{program} = \frac{time}{cycle} \cdot \frac{cycles}{instruction} \cdot \frac{instructions}{program}$$

CISC CPUs keep the number of instructions per program low, while the time/cycle is high. Because RISC CPUs do not have complex instructions, the number of instructions per

[18] tst, beq, bne, move, cmp, cmpi

program is higher, but this is compensated by a lower number of cycles per instruction. In practice, 32 bit PowerPC code is less than 10% larger than 32 bit i386 compiled code[19], but the fact that today's compilers generate RISC-like code also for CISC CPUs must be taken into account, so CISC code could be a lot smaller.

2.4.2 History

In 1990, IBM releases the RS/6000 (RS stands for "RISC System", later renamed to "IBM pSeries") UNIX workstation based on their new "POWER" ("Performance Optimization With Enhanced RISC") multi-chip CPU, which they would like to see being used in the desktop market as well [62], PowerPC-arstechnica. At the same time, Apple is looking for a new CPU for their Macintosh computer line, as they want to switch away from the Motorola M68K CISC architecture which has not been able to keep up with other architectures on the market. Motorola has released the M88K RISC CPU in 1988, but the market showed little interest.

So Apple, IBM and Motorola ("AIM") collaborate to design a CPU architecture. IBM contributes the instruction set and the design of the POWER CPU, Motorola contributes the hardware interface as well as the memory management and cache system of the M88K, and Apple contributes compiler construction knowledge used to refine the instruction set. In 1991, the "PowerPC Architecture Definition" (for 32 and 64 bit processors) is released. "Architecture Definition" means the exact software specification of the minimum functionality of a PowerPC CPU, independent of a possible implementation, as it has been done with the SPARC and MIPS architectures before. In 1992, the MPC601, the first PowerPC CPU is released, which is a hybrid between the POWER and PowerPC architectures that implements some instructions that were unique to the POWER architecture (the remaining ones could be trapped and emulated in software) in addition to being fully PowerPC compliant, so that it can be used in RS/6000 machines as well. In 1994, the first Macintosh with a PowerPC processor is released, the Power Macintosh 6100, introducing Motorola's PowerPC 601.

Aside from Macintosh systems, The PowerPC has since become very important in the embedded market, especially in automotive and multimedia applications. The Nintendo GameCube gaming console is driven by a PowerPC 750 (G3) by IBM, and the successor to Microsoft's Xbox will include three 64 bit IBM PowerPC 970 (G5) cores.

Since the formal definition of the PowerPC architecture, it has hardly changed. The definition already included 32 and 64 implementations, and a 64 bit PowerPC has not been available until 2003 with the IBM PowerPC 970 (G5). The only addition was the AltiVec (called "VelocityEngine" by Apple) SIMD unit, which was introduced with Motorola's G4 CPUs in 1999.

In the following chapters, only 32 bit implementation of the PowerPC without the floating point and AltiVec units will be of importance.

[19] "/bin/bash" of OpenDarwin 7: the PowerPC code is 598048 bytes, the i386 part 552891 bytes (8.2%); "/usr/bin/unzip": the PowerPC code (Mandrake Linux 9.1) is 111500 bytes, the i386 code (SuSE Linux 8.2) is 109692 bytes (1.6%)

2.4.3 Registers

The PowerPC has 32 general purpose registers, r0 to r31, each of them 32 bits wide. All of these registers can equally be used for logical and arithmetic operations. There is one exception to this: Some instructions cannot use r0 as a source, and use the encoding for an operand of the value ”0” instead. Furthermore, there are eight 4 bit condition registers cr0 to cr7, which are typically written to by compare instructions and evaluated by conditional jumps.
The PowerPC has two more special purpose registers: In case of a function call, the link register ”lr” gets set to the address after the call instruction, and the return instruction just jumps to the address stored in lr. This eliminates two stack accesses for the leaf function in a call tree. The count register ctr is mostly used in conditional jumps: The conditional jump instruction can optionally decrement the count register and make the branch additionally dependent of whether ctr reached zero. ctr is also used for calculated jumps by copying the address into ctr and using the ”bctr” instruction.
Unlike on the i386, user and supervisor registers are strictly separated, which makes the PowerPC easy to virtualize, and also simplifies recompilation of user mode code.

2.4.4 Instruction Set and Addressing Modes

The instruction set of the PowerPC consists of about 200 instructions. This sounds a lot, but since instructions and addressing modes cannot be strictly separated on RISC CPUs, all different formats of one operation count as individual instructions. Combining all instructions with all addressing modes on the i386, for example, would lead to a number above 1100 instructions[20]. However, the number of instruction the PowerPC has is still higher than average for a RISC CPU. All instructions are encoded in 32 bits, which is the reason for limits of certain instructions. As it is impossible to encode 32 bit constants, like on most RISC CPUs, immediate values can only be 16 bit (a load immediate of a 32 bit value must thus be done in two steps), for example, and jumps and branches only have a limited range.
Being a RISC, the PowerPC is a load/store architecture, which means that calculations can only be performed in registers, not in memory. Memory accesses are only possible for loading and storing data. The only complex addressing modes are possible with loads and stores: Indexed addressing with a displacement is supported. There are also load and store operations of multiple registers, and an indexed store instruction with displacement that can update the index register with the effective address.
Due to the relatively wide instruction code, the PowerPC can encode three registers in one instruction, so three register logic is possible, i.e. there can be two source registers and a target register for binary operations like ”add”. All binary arithmetic instructions support three register logic, and all unary instructions support two registers, source and target. All arithmetic and logic operands can be any of the 32 general purpose registers. Arithmetic

[20] Using Intel’s official i386 reference ”386intel.txt” [35], the UNIX command line `grep ^[0-9A-F][0-9A-F] .*" 386intel.txt | wc -l` (returns 585) is a rough estimate for the number of possible opcodes. `grep ^[0-9A-F][0-9A-F] .*/r" 386intel.txt | wc -l` (returns 91) gives an estimate of instructions that support ”r/m” addressing, and are thus available in 7 different addressing modes.

and logic instructions can optionally write information about the result (positive, negative, zero, overflow) into the first condition register (cr0). This is indicated by the suffix "." of the mnemonic. The fact that this is optional is handy for both the hardware implementation of the CPU (less dependencies in the pipeline) and for the implementation of an emulator (less unnecessary work).

The compare instructions can write the result in any of the eight condition registers. There is a compare instruction for signed and for unsigned data, so that the condition register will contain the precise information about the result instead of flags like carry, sign and zero, i.e. the condition register does not have to be interpreted later, and the conditional jump instruction need not be aware of whether the compared values have been signed or unsigned. These conditional jump instructions can make the jump depend on either of the 32 (eight times four) condition register bits and/or whether the counter register ctr, which can optionally be decremented by the conditional jump instruction, has reached zero.

2.4.5 Instruction Encoding

As already mentioned, all PowerPC instructions are encoded into a 32 bit value. The upper 6 bits form the primary opcode. Depending on the value of the primary opcode, there can be an extended opcode in bits[21] 1 to 9 or 1 to 10. The remaining bits encode registers, immediate values or additional flags. As mentioned earlier, the power and flexibility of an instruction is always based on how many operands and options fit into the instruction code. PowerPC instructions therefore often contain additional optional functionality or can encode different operations, so sometimes different mnemonics exist that lead to the same opcode, but with different flags set. The same is done if certain combinations of operands have a simpler function. For example, "addi r3, 0, value" is the same as "li r3, value", because r0 (written as "0" in this instruction) will always behave like the constant of zero in "addi".

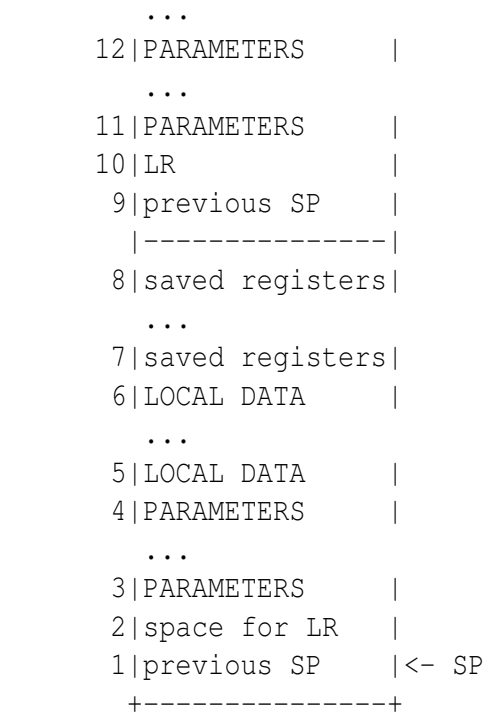
2.4.6 Endianness

The PowerPC CPUs can work both in big endian and in little endian mode, big endian being the native mode though. This means that data storage works in big endian mode by default, and a little endian mode can be simulated. The CPU does not reverse the byte order of all data when accessing memory, but leaves word accesses unchanged and only changes the effective memory address in case of non-word memory accesses. This behavior is transparent to the user, but it is not fully compatible with a real little endian mode when accessing memory mapped hardware.

All major PowerPC operating systems (Mac OS/Mac OS X, Linux, Nintendo GameCube OS, Xbox 2 OS) operate the PowerPC in big endian mode.

[21] This is little endian bit notation, i.e. bit 0 is the LSB

Figure 2.14: PowerPC Stack Frame



2.4.7 Stack Frames and Calling Conventions

Stack frames and calling conventions on the PowerPC are quite complex. Apple's description [34, pp 31ff] occupies six pages, for example.

As the general purpose register r0 is somewhat special in that it cannot be used as an index, the next register, r1, is used as the stack pointer by practically all operating systems, although any other register other than r0 could be used. The stack grows from bigger addresses to lower addresses. The PowerPC "EABI" ("Embedded Application Binary Interface") standard [33] also defines the layout of the stack and calling conventions, that is, how to pass parameters and return values.

Inside a function, the stack looks like figure 2.14 (bigger addresses on top). The stack pointer points to an address (1) that contains the previous stack pointer, in this case 9. The address above (2) is empty and can be used by a subfunction called by this function to save the link register. So this function saved the link register at address 10. At the top of the stack frame (7 to 8) volatile registers that the function modifies are saved. Below, local variables are stored (5 to 6) and just as much space is reserved to fit all parameters accepted by the subfunction with most parameters (3 to 4). The parameters for this function are stored in the caller's stack frame (11 to 12).

Although there is space for all parameters on the stack, most parameters are passed in registers: If the parameter data is less than 8 times 32 bits, the parameters are stored in r3

to r10[22]. If there are more parameters, the caller stores them in its stack frame leaving the first eight addresses (for r3 to r10) empty.
Results up to 32 bits are returned in r3, 64 bit results in r3 and r4, and larger results are written to memory pointed to by the implicit parameter r3. The stack pointer (r1), r13-r31 and cr2-cr4 must be preserved during a function call, all other GPRs, condition registers as well as the count register (ctr) can be overwritten. Due to the nature of function calls on the PowerPC, the link register will always be overwritten by the function call instruction.
Typical code to create and destroy a stack frame looks like this:

```
00001f5c    mfspr r0,lr         ; get link register
00001f60    stmw r30,0xfff8(r1) ; save r30 and r31 on stack
00001f64    stw r0,0x8(r1)      ; save link register into caller's SF
00001f68    stwu r1,0xffb0(r1)  ; stack pointer -= 0x30 and save it
[...actual code in the function...]
00001f80    lwz r1,0x0(r1)      ; restore stack pointer
00001f84    lwz r0,0x8(r1)      ; get link register from caller's SF
00001f88    mtspr lr,r0         ; restore link register
00001f8c    lmw r30,0xfff8(r1)  ; restore r30 and r31
00001f90    blr                 ; return
```

2.4.8 Unique PowerPC Characteristics

All RISC CPUs are very similar. They all have many general purpose registers, a fixed instruction length, few composite instructions and few addressing modes, so describing a RISC CPU is as easy as describing the differences to standard RISC paradigms.
The PowerPC, however, is no typical RISC. The POWER instruction set on which the PowerPC is based has been released in 1990, five years after the SPARC, MIPS and ARM, and therefore has some post-RISC features [31, pp C-18f]. It has more instructions than a pure RISC CPU, including some complex instructions like load/store multiple (handy to save/restore multiple registers on the stack), load/store with update (can be used to modify the stack pointer after a read or write access) and branch instructions that involve the count register. The concept of multiple condition registers is also untypical. Most RISC CPUs either have no condition register for compares at all (instead, comparison results will be stored in general purpose registers, like on MIPS and Alpha) or a single set of flags (like on ARM).
Also, the PowerPC's connection to the bus is quite flexible. Unlike most other RISC CPUs, it allows misaligned memory accesses, and it can switch the endianness[23].

[22] The rules are somewhat more complicated when floating point values are passed.

[23] The IBM PowerPC 970 (G5) lacks the feature to switch the endianness, which made VirtualPC 6 incompatible with the new Apple Macintosh line.

Chapter 3

Design

This chapter describes the ideas, concepts and the concrete architecture of the PowerPC to i386 recompiler that has been developed in the context of this thesis.

3.1 Differences between PowerPC and i386

In order to maximize the performance potential of a recompiler, having a close look at the concrete differences between the source and the target architecture is very important.

3.1.1 Modern RISC and CISC CPUs

RISC/CISC is often regarded as *the* difference in CPU design, and it is a very widespread view that CISC is inferior to RISC in general, so modern CISC CPUs such as the Pentium 4 or the Athlon XP/64 must be inferior to the PowerPC G4/G5. But we are living in a post-RISC world [52]: RISC and CISC have moved towards each other.
When comparing modern RISC and CISC CPUs, the instruction set and the internals must be observed. The instruction sets of modern PowerPC and i386 CPUs are still very different. i386 CPUs have fewer registers (and therefore pass parameters using the stack), no link register, no three operand logic and complex composed instructions; and instruction encoding is very complex. But these characteristics do not necessarily describe the way the i386 is used: Modern compilers typically do not emit CISC-typical composed instructions (like ”loop” and ”enter”), for example, but emit several more simple (RISC-like) instructions instead. Even the more simple instructions are converted into more RISC-like internal instructions by the instruction decoder and the few i386 registers are mapped to more internal registers to minimize dependencies. Complex instructions, which should be rarely used by applications, are still interpreted by microcode [37], which is thus mainly used to maintain backwards compatibility, as complex instructions are not native to the CPU any more. So modern CISC CPUs could be viewed as ”RISC implementations of CISC architectures” [63].
Internally, modern PowerPC and i386 CPUs are very much alike [52], and very complex. They include complex caching techniques, and units that allow superscalar execution, out-of-order execution and branch prediction in order to improve performance. At the begin-

Table 3.1: Transistor counts of recent CPUs

CPU	Year	Transistors (million)
G4 (7451)	2001	33
Pentium 4 (Northwood)	2002	55
Athlon XP (Barton)	2002	54
G5 (970)	2003	52
Athlon 64	2004	68/105

ning, RISC CPUs have been a lot less complex than CISC CPUs: The first ARM CPU (RISC, 1986) only had 30,000 transistors, while the Motorola 68000 (CISC, 1979) had 68,000 - the RISC CPU had less than half the transistors, although it was seven years younger and a lot more powerful. Today, the differences have become smaller due to the increased importance of other units than the instruction decoder, as table 3.1 demonstrates (data taken from [64]).
The i386 CPUs need to support many legacy features and its instruction set is certainly less than optimal. The most important deficiency is the very low number of registers (only eight), but since 2004, the new lines of AMD and Intel i386 CPUs (AMD64/EM64T) have 16 general purpose registers - which is more than the RISC CPU ARM, which has one register less[1].
After all, the i386 is not so bad at all nowadays, as it is a lot closer to RISC CPUs like the PowerPC as it used to be. In practice, modern i386 CPUs achieve about the same performance as their PowerPC counterparts. In certain situations, CISC CPUs can even be faster. Hand optimized i386 code can sometimes encode an operation a lot more efficiently, either because a microcoded operation is still faster than a function in assembly in some cases (e.g. i386 memory copy ”movs”) or because more tricks are possible, as the instruction set is more complex (for example the multiplication by five shown earlier).
So modern RISC and CISC CPUs are very much alike, and both have their unique advantages. Sure, RISC is the cleaner design, but both designs have their right to exist. This is another motivation for targeting CISC with a recompiler.

3.1.2 Problems of RISC/CISC Recompilation

Although the inner workings of today’s desktop CPUs are very similar, regardless of whether they are RISC or CISC, the instruction set architecture and the conventions are very different. Recompiling code between similar architectures is easy and produces good and fast code. It is the differences that makes translation hard and the target code inefficient. These are the most important differences in the instruction set between the PowerPC and the i386:

- **Too few registers**: The PowerPC has 31/32 general purpose registers (r0 might not be regarded as fully general purpose), while the i386 only has 8, some of which are the only ones that can be used in certain combinations (multiplication, stack). Compiled PowerPC code always makes optimal use of the registers, often using more registers that would be necessary, in order to keep dependencies between instructions low.

[1] r15 is the program counter, which is a non-GPR on the ARM.

Mapping only some PowerPC registers to i386 registers and mapping the remaining ones to memory is generally slower than should be necessary.

- **Incompatible condition registers**: The PowerPC has eight condition registers, compared to the single one of the i386. In addition, they are incompatible: PowerPC condition codes store a concrete result (greater, lower, equal), while the i386 flags store data that has to be interpreted later (sign, overflow, carry, zero). So on the PowerPC, only the compare instruction has to be aware of whether the values are signed or unsigned, while on the i386, only the conditional jump has to be aware of it. Explicitly calculating flags can be fatal for an emulator, as every source compare/branch combination (which are expensive by definition) may lead to several additional compare/branch instructions on the i386, in order to manually test the result. Furthermore, the PowerPC only updates the condition codes after an arithmetic instruction if a bit in the instruction code is set, while the i386 always updates the flags, so the i386 flags can hardly be used to store PowerPC condition codes, as they would be frequently overwritten.

- **Missing link register**: Unlike the PowerPC, the i386 has no link register. Any i386 register can take the place of a link register, though, because all it has to support is load, store, load immediate, and a jump to the address pointed to by the value of the register. Unfortunately this has drawbacks: i386 registers are few, and this would occupy yet another register, and jumps to locations pointed to by registers are slower on the i386 than a simple ”ret” instruction, as the latter case is very optimized: ”ret” instructions are detected very early so that the target of the jump will also be known very early to fill the pipeline in time, but jumps to values in registers are typically only handled by the branch prediction logic, so the pipeline might not be filled in time.

- **Different stack conventions**: The stack is very different on the PowerPC compared to the i386. First, there is no instruction that could be mapped to ”push”: While values are ”pushed” on the stack (store/sub/store/sub...) on the i386, they are stored (SP indexed with displacement) on the stack on the PowerPC, decreasing the stack pointer once afterwards (store/store/store/sub). The i386 can emulate this behavior, but it bloats the code. Second, the complete conventions are different, for example, the passing values to functions using the registers is not possible on the i386, instead they would be stored on the stack. As before, the i386 can emulate this behavior, but it is not optimal, as its ”native” stack structure would be a lot faster for many reasons, one of them being that indexing with EBP is faster than with ESP because of the shorter instruction encoding.

- **Calculated jumps**: Calculated jumps (function pointers, ”switch” constructs) are a problem no matter how similar the source and target architectures are. On any architecture, a sequence that performs a calculated jump ends with the actual jump instruction, which is hard to translate, as the address that is supposed to be jumped to is an address in the code of the source program, while the translated code must jump to an address in the target program. In static jumps, this can be easily translated, but

in calculated jumps, it cannot. This is worse when translating from the PowerPC to the i386: Address calculation is complicated and cannot be preformed in a single instruction, so the complete sequence would have to be detected and replaced with i386 code. Translating from the i386 to any architecture would have been simpler: One single instruction (something like `jmp *table_address(,%ebx,4)` is used) does the complete address calculation and can therefore be translated more easily.

- **Endianness**: In most applications, the PowerPC is big endian, while the i386 is always little endian. If all memory accesses always were aligned 32 bit accesses, this would be no problem, but the PowerPC supports byte and halfword accesses as well as unaligned accesses. Every (unaligned or non-word) access by the PowerPC would have to go along with endianness conversion on the i386.

3.2 Objectives and Design Fundamentals

Before starting with the actual design of the system, the precise objectives of it have to be known, especially in areas that are supposed to differ from what other solutions currently achieve. As some objectives are mutually exclusive, this definition is not easy.

3.2.1 Possible Objectives

There are three main objectives an emulator can have:

- high compatibility
- low latency
- high performance

On first sight, these three objectives seem incompatible. Low latency is mutually exclusive with high performance, because significant improvements in the quality and speed of the generated code can only be achieved by making the recompiler slower and thus increase the latency of execution. Similarly, high speed and high compatibility are incompatible: The higher the accuracy the slower the emulation, as shown earlier.

3.2.2 How to Reconcile all Aims

As it is true in many areas of engineering, there are methods that make all objectives possible to some extent, typically by making compromises and applying combinations of simpler methods.

- **High compatibility**: In this case, high compatibility can be regarded as a matter of definition: If the goal is to achieve high compatibility, but only doing user mode instead of system emulation, then the recompiler only needs to be basic block accurate. There is no need to emulate exact interrupts or the MMU of the CPU, which makes

emulation that is a lot faster possible. For high compatibility really to be achieved, a dynamic recompiler must be used instead of a static one, and a fallback interpreter must be included for calculated jumps and jump tables.

- **Low latency**: A recompiler that produces good code slows down latency. Therefore the hotspot principle must be applied: An emulation method with low latency will be used for most of the code, and hot spots will be optimized.

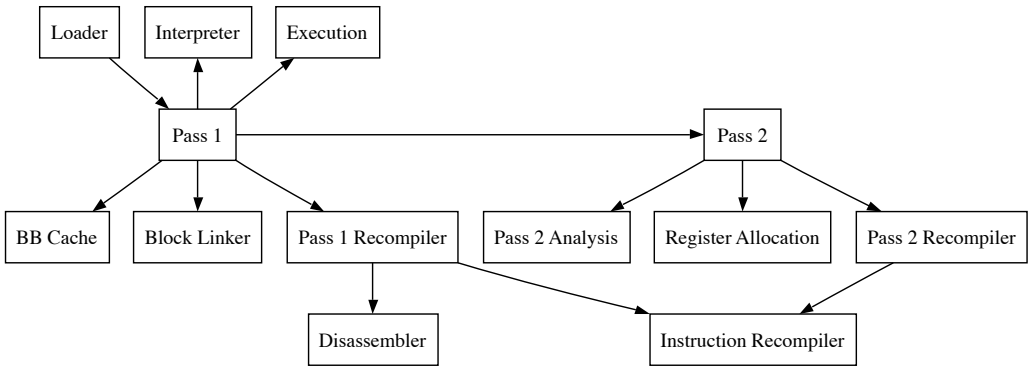
- **High speed**: High speed can be gained by a recompiler that produces very good code. Of course this can only be done for hotspots, in order not to increase latency. This ”make the common case fast” principe can also be applied in a second way: Today’s code is typically compiled code, not hand optimized assembly. Furthermore, as only user mode emulation has to be done, operating system code like mode switches do not have to be coped with. Therefore the system is supposed to be optimized for machine code that has been produced by a compiler. Since compilers produce some typical sequences, additional optimizations can be done.

Another idea to improve the performance of the generated code is to make maximum use of the characteristics of the target CPU. Emitting good target code need not be slower than emitting bad code. A rule of thumb for good target code is: If the produced code looks like code that would be emitted by a standard compiler, then the code is fast, as CPUs are optimized for compiled output as well.

3.3 Design Details

The basic idea of the system is hotspot recompilation, but with fast recompilation pass instead of interpretation for non-hotspot code. Pass 1, the fast recompiler, is a dynamic recompiler doing static register allocation (six PowerPC registers are mapped to i386 registers, the rest is mapped to memory) and translating every instruction independently, without an intermediate language. The units of code managed by pass 1 are basic blocks, which are translated one by one and cached after translation. After every execution of a basic block, the recompiled code jumps back to the recompiler, unless the following basic block or blocks are known, in which case the recompiler links the blocks together, making the jump back to the recompiler obsolete.
The pass 2 recompiler, which is applied on hotspots, makes use of more sophisticated optimization techniques. It always works on whole functions, makes register usage statistics and uses dynamic register allocation to optimize register usage on the i386. Additionally, PowerPC-style function calls (link register) are mapped to i386-style calls (call/ret), and stack frames are optimized.
The loader analyzes the executable file that is to be emulated, loads its sections into memory and passes all data to pass 1 logic, which is the core of pass 1 recompilation. Pass 1 logic decides about the sizes of sequences that will be recompiled, passes basic blocks of code to the actual recompiler, recompiled code to execution and individual instructions to the interpreter. It decides when to link basic blocks as well as when to hand code to pass 2. The basic block cache maintains an archive of basic blocks that have already been recompiled.

Figure 3.1: Design of the System



The pass 1 recompiler passes every single instruction to the instruction recompiler and, optionally, to the disassembler for debugging purposes. The instruction recompiler translates a single PowerPC instruction into i386 code; PowerPC registers get mapped to i386 registers or memory locations by the register mapper.
Pass 2 logic is the core of pass 2 recompilation. It finds whole functions, passes them to the register allocator in order to get the optimal register allocation for every function, and then passes them to the pass 2 recompiler. With the help of pass 2 analysis, register usage statistics are made, then liveness analysis is done and finally an optimal allocation is found. The pass 2 recompiler passes every single instruction to the instruction recompiler, just as in pass 1, but this time the instruction recompiler is in a different mode so that it will use the dynamic register allocation and optimize stack accesses.
Some of the following sections might not always describe some design accurately at once. Instead, only the aspects needed for a certain functionality are described, and additions that are needed for another functionality are described later. This approach has been chosen to make the design easier to understand, although it makes it harder to make practical use of the information without reading to the end.

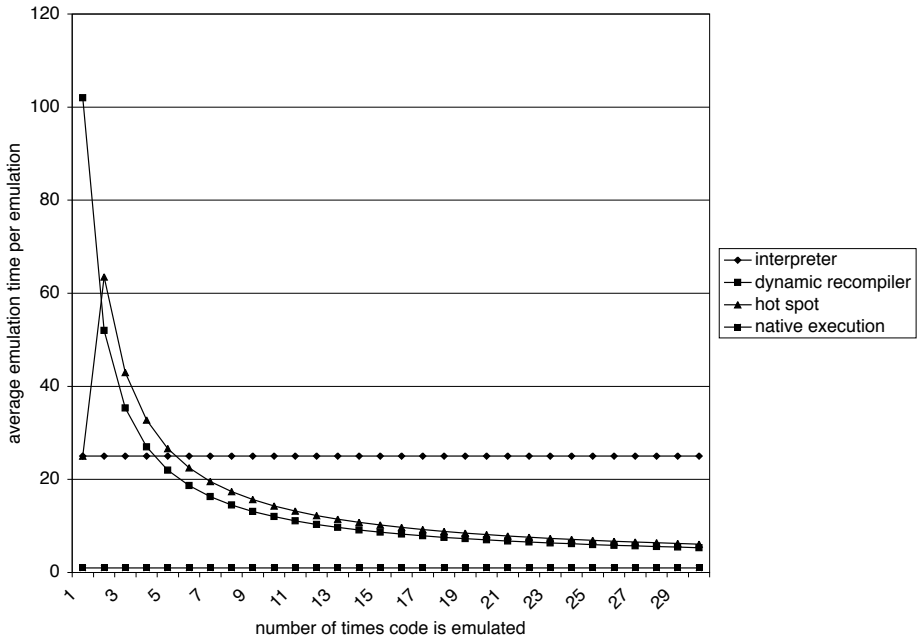
3.3.1 The Hotspot Principle Revisited

The key to low latency and high speed is the hotspot principle. A lot of code is only executed once. It makes no sense to translate this code, as translation is a lot slower than a single interpreted execution. The average costs of a recompiled execution are

$$\frac{cost\_of\_recompilation}{number\_of\_times} + cost\_of\_execution$$

This results in high costs if executing only a few times, given recompilation is slow. The hotspot method eliminates the peak at the beginning by interpreting at the beginning and recompiling later. For few runs this is faster than recompilation, and for many runs this is just a little slower than recompilation.
But the two assumptions that have been made are not entirely accurate. Recompilation is not necessarily much slower than interpretation. A recompiler optimized for compilation speed might be only slightly slower than an interpreter that has been optimized for speed.

Figure 3.2: Speed of the Traditional Hotspot Method



Also, while it is true that a lot of code only gets executed once per run of an application, most applications are run multiple times, and many of them, such as UNIX tools, very often. If we do not count a single machine, but a complete network of machines, this is even more true.

As it has been shown earlier, the part that makes an interpreter so slow is the dispatcher jump or jumps and decoding. The example needed 28 cycles for the interpretation of one instruction, 15 of which are caused by the unpredictable jump. The following code is based on the interpreter, but it has been changed so that the instruction does not get executed, but i386 code gets emitted instead that matches the PowerPC instruction:

```
  1 mov (%esi), %eax              // fetch
  2 add $4, %esi
  3 mov %eax, %ebx                // dispatch
  4 and $0xfc000000, %ebx
  5 cmp $0x7c000000, %ebx
  6 jne opcode_1_is_not_31
  7 mov %eax, %ebx
  8 and $0x000007fe, %ebx
  9 jmp *dispatch_table(,%ebx,4)
--------------------------------
 10 mov %eax, %ebx                // decode
 11 shr $21, %ebx
 12 and $31, %ebx
 13 mov %eax, %ecx
```

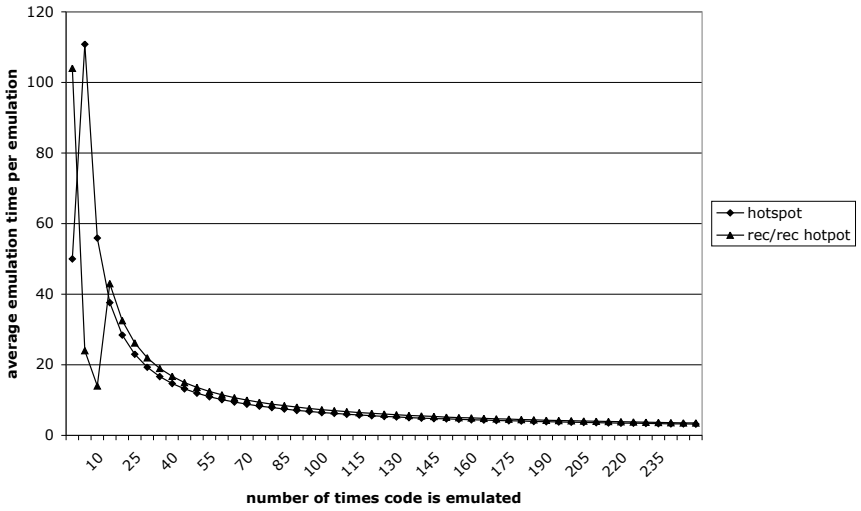
```
14 shr $16, %ecx
15 and $31, %ecx
16 mov %eax, %edx
17 shr $11, %edx
18 and $31, %edx
19 movl mapping(,%ebx,4), %eax  // emit i386 code
20 movb $0xa1, (%edi)
21 movl %eax, 1(%edi)
22 movl mapping(,%edx,4), %eax
23 movw $0x050b, 5(%edi)
24 movl %eax, 7(%edi)
25 movl mapping(,%ecx,4), %eax
26 movb $0xa3, 11(%edi)
27 movl %eax, 12(%edi)
28 add $16, %edi
29 jmp interpreter_loop          // loop
```

While 3 instructions were necessary for the execution, the recompiling part in this example consists of 10 instructions for the translation (instructions 19 to 28). One iteration of the interpreter took 28 cycles - this recompiler code takes 34 (see Appendix A). While the chart above assumed that recompilation was slower by a factor of 4, in this case, recompilation is only 26% slower than interpretation.
The recompiler can be this fast because it does not do any optimizations at all and maps all PowerPC registers to memory locations. ”mapping”, in this case, is a 32 entry table that contains the addresses in memory used for register emulation. Of course, the code produced by this recompiler is not optimal:

```
a1 00 00 01 00          mov    0x00010000,%eax
0b 05 04 00 01 00       or     0x00010004,%eax
a3 08 00 01 00          mov    %eax,0x00010008
```

This code might be slower by a factor of 3 or 4 than the output of a more optimized recompiler that uses i386 registers. It is certainly possible to improve the quality of the output code, but this will slow down the recompiler.
A compromise has to be found between recompilation and execution speed. A sensible goal is a recompiler that consumes less than double the time of the interpreter. The produced code should be slightly better than the example code above.
Eliminating the interpreter (except for certain jump constructs) has two important advantages: There is less code in memory during execution, and there is a lot less source code to maintain. A system that includes a full interpreter and a full recompiler has roughly double the lines of the recompiler alone.
Of course the code produced by a recompiler that has double the costs of an interpreter emits better code than shown above, but the code is probably still far from being optimal. Therefore it makes sense to apply the hotspot method, by combining a fast and an optimizing recompiler. Instead of the interpreter, a fast recompiler already translates the

Figure 3.3: Speed of the Recompiler/Recompiler Hotspot Method



$c_i = 50$, $c_{e1} = 4$, $c_{r1} = 100$, $c_r = c_{r2} = 500$, $c_e = c_{e2} = 1$, $t = 1$, $t_{rr} = 10$

PowerPC code when a block is first encountered. Subsequent runs are done by executing the recompiled code. After a certain number of times a block of code has been run, it will be recompiled by an optimizing recompiler, which is of course a lot slower than the fast recompiler, but it also produces a lot better code.

The following graph compares the traditional hotspot method with the recompiler/recompiler hotspot method:

These are the formulas that have been used:

Hotspot Method	if $n > t$ then $tc_i + c_r + (n-t)c_e$ else nc_i
Rec/Rec Hotspot Method	if $n > t_{rr}$ then $c_{r1} + nc_{e1} + c_{r2} + (t_{rr} - n)c_{e2}$ else $c_{r1} + nc_{e1}$

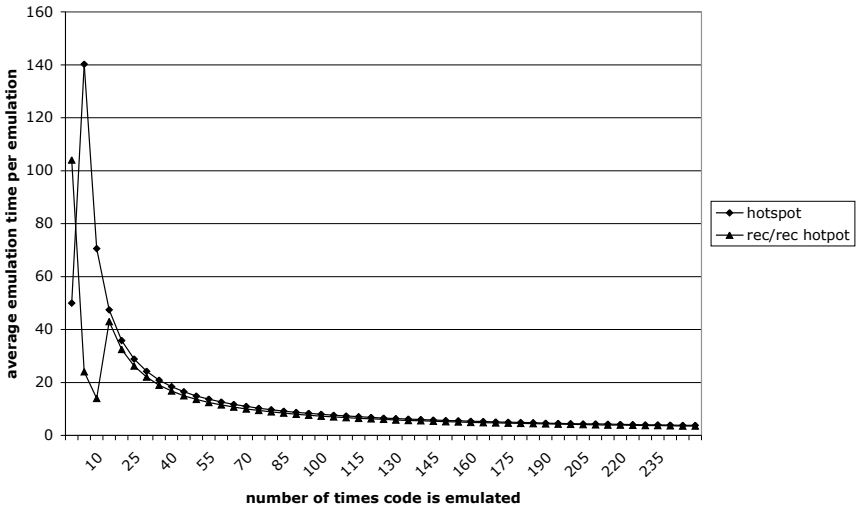
c_{r1} and c_{r2} are the costs of pass 1 and pass 2 recompilation. c_{e1} and c_{e2} are the costs of pass 1 and pass 2 execution. t_{rr} is the threshold for pass 2 recompilation.

In the example shown in figure 3.3, the traditional hotspot method is of course faster by a factor of 2 when executing an instruction once, because it interprets the instruction. Up to the threshold of the recompiler/recompiler hotspot method, the recompiler/recompiler method is faster, because the slower recompilation of the traditional method has not amortized yet. The second pass of the recompiler/recompiler method makes this method slower, but only slightly. After 100 runs, they are practically equally fast.

There are a lot of numbers to play around with: If a lot of code runs up to four times, it makes sense to set the threshold of the traditional hotspot method to something higher, like five. In this case, the recompiler/recompiler method is faster in all cases other than 1 (figure 3.4).

In oder to achieve a translation time that is less than twice the interpretation time, there is

Figure 3.4: Speed of the Recompiler/Recompiler Hotspot Method (different parameters)



$c_i = 50$, $c_{e1} = 4$, $c_{r1} = 100$, $c_r = c_{r2} = 500$, $c_e = c_{e2} = 1$, $t = 4$, $t_{rr} = 10$

little choice in methods and algorithms. As the complexity of the interpreter is $O(1)$, this means that the complexity of the recompiler must be $O(1)$ as well, so all instructions must be translated separately.

3.3.2 Register Mapping

If the destination CPU has a lot more registers than the source CPU, source registers can be simply mapped to target registers. If this is not the case, like in the case of PowerPC to i386 recompilation, it gets more difficult. High level language compilers have the same problem: Local variables are supposed to be mapped to registers. This is done by using liveness analysis and register allocation algorithms on functions. Unfortunately, these algorithms are quite slow: The complexity is between $O(1)$ and $O(n)$ per instruction, n being the number of instructions within the function. Therefore the standard register allocation algorithm is not an option for pass 1.

3.3.2.1 Simple Candidates

There are several simple methods that have a complexity of $O(1)$.

Memory Mapping The most simple method is of course the ”do nothing” method. Instead of mapping some source registers to target registers, all source registers are mapped

to memory. An array in memory contains all registers, and every register access in the source language will lead to a memory access in the recompiled code.
”Do nothing” is the fastest method in terms of recompilation speed. All that has to be done is read the register-to-memory mapping from a table and emit opcodes that have memory locations as operands. The sequence that does the actual translation does not contain a single conditional jump:

```
movl mapping(,%ebx,4), %eax  // emit i386 code
movb $0xa1, (%edi)
movl %eax, 1(%edi)
movl mapping(,%edx,4), %eax
movw $0x050b, 5(%edi)
movl %eax, 7(%edi)
movl mapping(,%ecx,4), %eax
movb $0xa3, 11(%edi)
movl %eax, 12(%edi)
add $16, %edi
```

Of course, the ”do nothing” method produces very inefficient code, because every register access in the source language will be translated into a memory access in the recompiled code. The example code that is supposed to translate ”or r1, r2, r3” into i386 assembly, would produce the follwing code:

```
movl register+2*4, %eax
orl register+3*4, %eax
movl %eax, register+1*4
```

Modern CPUs can do one memory access in every three clock cycles without being delayed[37], so this means that there may be one memory access about every six instructions, in practice. This solution has a memory access in nearly every instruction, so it might be about five times slower than the same code using registers instead:

```
movl %ebx, %eax
orl %ecx, %eax
```

Static Allocation Static register allocation means that a hardcoded set of source registers will be mapped to target registers, and the remaining registers will be mapped to memory. The recompiler will be slower, of course. In a naive implementation it would check every operand register whether it is mapped to a register or to memory, leading to one conditional jump per operand:

```
mov mapping(%eax), %eax
cmp $8, %eax
jb case_rxx
[...]
-----------------------------
```

```
case_rxx:
    mov mapping(%ebx), %ebx
    cmp $8, %ebx
    jb case_rrx
    [...]
----------------------------
case_rrx:
    mov mapping(%ecx), %ecx
    cmp $8, %ecx
    jb case_rrr
    [...]
----------------------------
case_rrr:
    [...]
```

Three register operands, each of which is either register or memory mapped, lead to 8 combinations: The code has to test one operand register after another in order to reach the correct one of the eight cases. Since not all instructions have three operands, the average number of non-predictable conditional jumps per instruction is probably around 2.
Using a trick, an implementation in assembly can reduce this to one single non-predictable jump:

```
xor %edx, %edx

    mov mapping(%eax), %eax
    cmp $8, %eax
    rcl %edx, 1

    mov mapping(%ebx), %ebx
    cmp $8, %ebx
    rcl %edx, 1

    mov mapping(%ecx), %ecx
    cmp $8, %ecx
    rcl %edx, 1

    jmp *jumptable(,%edx,4)

jumptable:
.long case_mmm, case_mmr, case_mrm, case_mrr
    .long case_rmm, case_rmr, case_rrm, case_rrr
```

The information whether a source register is register or memory mapped is converted into one bit (”cmp”), and the three bits are combined in one register (”rcl”), so that the resulting number of 0 to 7 represents the combination. Using an eight entries jump table, the correct case can be reached using just one jump.

Certainly no C compiler can do this kind of optimization, therefore a C implementation cannot achieve this performance. Though, it makes sense to count the lowest possible costs, as the implementation might as well be written in assembly.

LRU The LRU ("least recently used") algorithm is an adaptive method. At the beginning of every basic block, all registers are stored in an array in memory, and all target registers are empty, i.e no source register is mapped to a target register. If an instruction that has to be translated needs to write a value into a source register, which is not mapped to a target register, the recompiler has to do the following:

- Check whether there is a free target register.
- If there is one, map the source register to this target register.
- If no target register is free, take the target register that has been least recently used. Emit code to write the old contents to the representation of the register in memory, and map the new source register to this target register.

If an instruction needs to read a register, the same will be done, and code will be emitted that reads the source register's representation in memory into the target register. At the end of a basic block, code must be emitted that writes all mapped registers that have been modified back into memory.
The LRU idea is certainly a good one, as it makes sure that registers that are extensively used within a sequence of code are mapped to target registers. The main problem of this method is its dependency on the size of basic blocks. Basic blocks in the typical definition are quite small:

```
-------------------------------------
 00001eb0        cmpwi   r3,0x1
 00001eb4        or      r0,r3,r3
 00001eb8        ble     0x1ee4
-------------------------------------
 00001ebc        addic.  r3,r3,0xffff
 00001ec0        li      r9,0x0
 00001ec4        li      r0,0x1
 00001ec8        beq     0x1ee0
-------------------------------------
 00001ecc        mtspr   ctr,r3
-------------------------------------
 00001ed0        add     r2,r9,r0
 00001ed4        or      r9,r0,r0
 00001ed8        or      r0,r2,r2
 00001edc        bdnz    0x1ed0
-------------------------------------
 00001ee0        or      r0,r2,r2
-------------------------------------
```

```
 00001ee4        or      r3,r0,r0
 00001ee8        blr
-------------------------------------
```

This iterative Fibonacci implementation, taken from 2.1.4.2, consists of six basic blocks. The average basic block size in this example is 2.5 instructions. This may not be representative, but basic blocks are rarely longer than 10 instructions, especially within loops, when it would be most important [50]. This would mean that this method is barely faster than the ”do nothing” method, as in practice, most registers would be read, updated and written again:

```
a:  mov register+3*4, %eax // EAX = r3
    cmp $3, %eax
    mov %eax, %ebx         // EBX = r0
    mov %ebx, register+0*4 // write back EBX
    jle f
-------------------------------------------
b:  mov register+3*4, %eax // EAX = r3
    dec %eax
    mov $0, %ebx           // EBX = r9
    mov $1, %ecx           // ECX = r10
    mov %eax, register+3*4 // write back EAX
    mov %ebx, register+9*4 // write back EBX
    mov %ecx, register+10*4// write back ECX
    je e
-------------------------------------------
c:  mov register+3*4, %eax // EAX = r3
    mov %eax, %ebx         // EBX = ctr
    mov %ebx, register+33*4// write back ctr
-------------------------------------------
d:  mov register+9*4, %eax // EAX = r9
    mov register+0*4, %ebx // EBX = r0
    mov %eax, %ecx         // ECX = r2
    add %ebx, %ecx
    mov %ebx, %eax
    mov %ecx, %ebx
    mov register+33*4, %edx// EDX = ctr
    dec %edx
    mov %eax, register+9*4 // write back EAX
    mov %ebx, register+0*4 // write back EBX
    mov %ecx, register+2*4 // write back ECX
    mov %edx, register+33*4// write back EDX
    jne d
-------------------------------------------
e:  mov register+2*4, %eax // EAX = r2
```

Table 3.2: Speed comparison of the three register allocation methods (estimated average clock cycles)

	memory	static	LRU
recompiler	10	30	50+
code	10	2	1-5

```
    mov %eax, %ebx          // EBX = r0
    mov %ebx, register+0*4 // write back EBX
-----------------------------------------
f:  mov register+0*4, %eax // EAX = r0
    mov %eax, %ebx          // EBX = r3
    mov %ebx, register+3*4 // write back EBX
    ret
-----------------------------------------
```

As a solution, blocks would have to be bigger. In this example, the whole algorithm could be regarded as a single block. The benefit would be register mapped registers within all loops. Unfortunately, if blocks are bigger than basic blocks, the mapping from source to target code addresses has to be managed at this level already. Also, it is more likely that it is found out after the translation that an instruction within the block can be the target of a jump from outside, so the block or at least a part of it has to be translated again. And finally, while it is known that basic blocks will always be executed completely, translating bigger blocks may lead to translated code that will never be executed.
Estimating the time needed per instruction for this algorithm is more complicated. For every operand register a non-predictable check has to be made whether it is already mapped. These three checks can be combined into a single check, as before. If the source register is not mapped, the first free target register has to be found. This loop can be unrolled, but will always consist of one non-predictable jump. If no target register is free, the oldest time stamp has to be found in another loop. Again, this loop can be unrolled, and by using the i386 conditional move instruction (”cmov”), no conditional jump is necessary.
So the recompilation of one instruction will need at least one branch to find out the mapping situation. One more branch is needed for every first use of a source register. In case many registers are used, the amount of code that has to be executed for the algorithm might also be significant.
As said before, the quality of the emitted code depends a lot on the size of the blocks. In case of small blocks, code quality is barely better than when doing memory mapping, but it can be very good when using very big blocks. With bigger blocks, code quality is especially good for loops. No matter what registers are used within the loop, they can be practically always register mapped. But the bigger the loops are, the more registers are used, the slower the algorithm will be.

Conclusion Table 3.2 summarizes the estimated average amount of cycles spent for recompiling an instruction using one of the three methods described above:

Table 3.3: Speed comparison of the three register allocation methods, dispatching and decoding included (estimated average clock cycles)

	memory	static	LRU
recompiler	35	55	75+
code	10	2	1-5

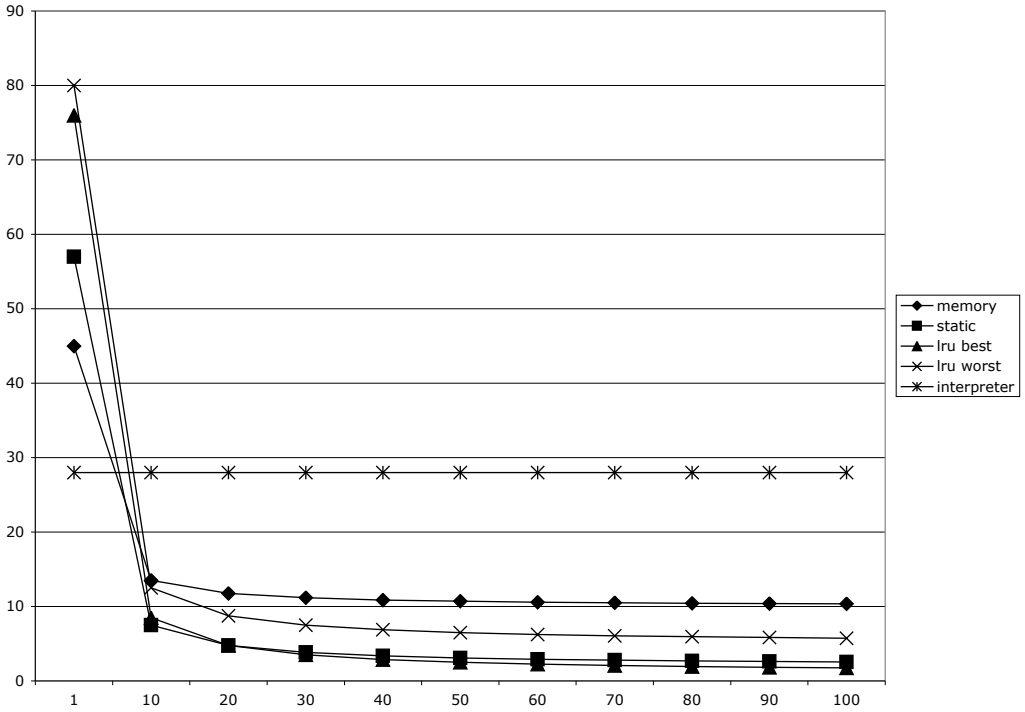
The ”do nothing” (memory mapping) method consists of few instructions, but both the recompiler and the produced code include three memory accesses, so they will take about 10 cycles each on a modern CPU. The static method consists of little more code and has three memory accesses, but includes one branch instruction, which will lead to about 30 cycles for the recompiler. Given 50% of all source register accesses can be mapped to target registers and an instruction consists of two register operands on average, 50% of the produced sequences should execute in about one cycle, and the other 50% in three cycles, as they require a memory access, leading to an average of 2 cycles.
It is a lot more complicated to give numbers for the LRU method. The recompiler has three memory accesses and one branch as well. If every instruction has two register operands on average and for every register access, the probability is 50% that the register is not mapped, then one register has to be mapped per instruction, adding the (optionally unrolled) search loop and another branch. 50 clock cycles is an optimistic estimate which will only be true if few registers are used, blocks are small and the emitted code is not optimal and might need more than 5 cycles to execute. In case of bigger blocks and more translation time, the code quality can be excellent, and one instruction might be executed in one cycle.
In order to be able to compare these values, 25 cycles have to be added. They represent the dispatcher and the decoder that are necessary in every iteration of the recompiler (table 3.3).
In this comparison, static register allocation is only slightly slower than memory mapping when code is executed once (127% of the costs), about just as fast when executing code twice (107%) and already faster when executing code three times (94%). After 1000 runs, the limit, static allocation being five times as fast, is practically reached. Compared to an interpreter (28 cycles per instruction), static allocation needs about twice as many cycles (196%), which matches the original objective.
On small blocks, LRU is slower and produces slow code. On big blocks, it produces code that is faster than when using the static method, but translation alone should take about twice as long in this case. LRU has a lot of potential, but it is inappropriate as a very fast algorithm.
As the static method is only slightly slower than memory mapping but produces clearly better code, this method has been chosen.

3.3.2.2 Details of Static Mapping

When doing static register allocation, there are several questions that have to be answered:

- How many target registers can be used for mapping source registers to? How many target registers are needed for other purposes?

Figure 3.5: Register Allocation Methods



- What source registers are supposed to be mapped? What are the most important source registers?
- Some PowerPC and i386 registers have a special purpose. Are there optimal target registers for certain source registers?
- How can the stack pointer and the link register be emulated - can they be treated like any other register?

Number of Mapped Registers The i386 has eight general purpose registers. EAX, EDX and ESP are not-so-general purpose: All multiplication and division instruction (except for ”imul”) use EAX and EDX as implicit parameters, and the stack pointer cannot be easily used as a register. At first sight, it would make sense to map source registers to all i386 registers other than EAX and EDX, which should be free for temporary operations, and to map the PowerPC stack pointer to ESP.
It has to be found out how many scratch i386 registers are needed to translate PowerPC instructions that use memory mapped registers. The ”or” instruction with its three operands, for example, needs one i386 scratch register, there is no possibility to do it without one:

```
mov register+0*4, %eax
or register+1*4, %eax
mov %eax, register+2*4
```

If the PowerPC instruction has three operands, and all of them are memory mapped, only one scratch register is necessary. But indexed memory accesses need up to two scratch registers. ”stw r10, 8(r10)” for example would be translated into this:

```
mov register+10*4, %eax
mov register+20*4, %ebx
mov %ebx, 8(%eax)
```

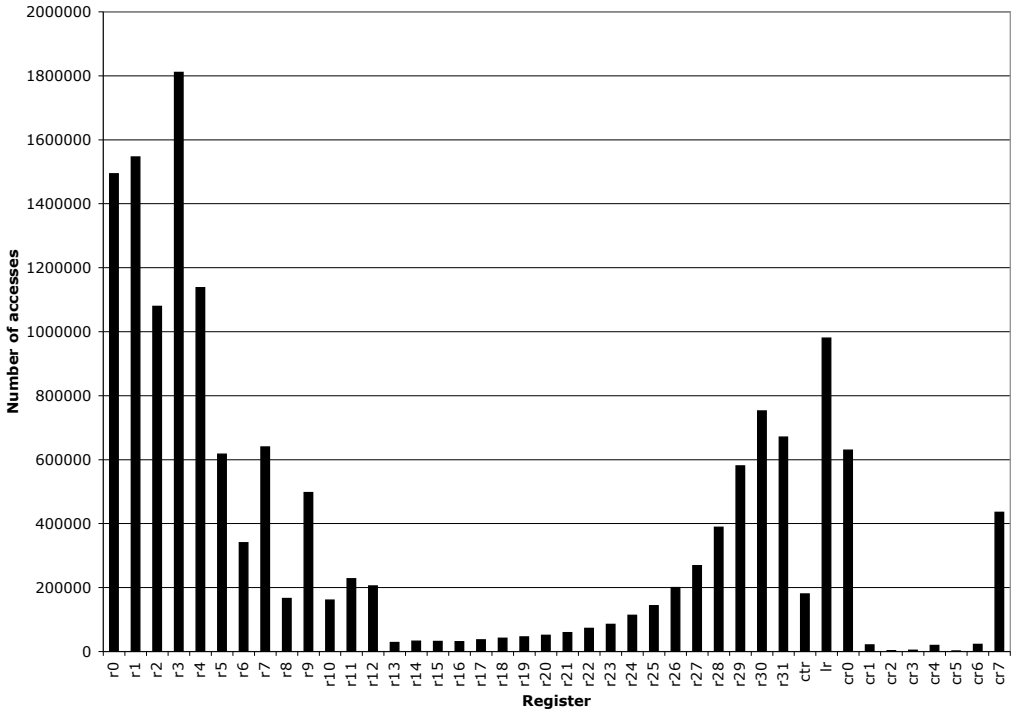
The last instruction needs two i386 registers, so two scratch registers are needed[2]. So it makes sense to use EAX and EDX as scratch registers.
The i386 stack pointer ESP cannot be easily used as something other than a stack pointer. In case of an interrupt/exception, or, in user mode, in case of a signal, ESP is always used as the stack pointer and data is written at the address ESP points to. So ESP could either not be used and always point to some writable location, or it could be used as the stack pointer. The PowerPC stack (typically) grows from higher to lower addresses, just like on the i386, so the stacks are basically compatible. Being a general purpose register, all instructions and addressing modes (except for the rather esoteric scaling of an index) are also possible with ESP. There is just one disadvantage: The encoding of ESP-indexed addressing modes are slightly longer than indexed addressing modes with other registers, but this should only be a cosmetic problem.

Mapped Source Registers Now that we know that we can map 5 source registers and the stack pointer to i386 registers, we need to know what source registers should be mapped. The performance of the recompiled code certainly depends on the choice of the registers. Unfortunately, compilers tend to use more registers than necessary. Compiled code that uses 20 variables, stores them in 20 registers, as there are enough registers, and this minimizes dependencies in the pipeline of the CPU. So in this case, only a maximum of 5 registers would be register mapped, and 15 registers would be memory mapped.
However, some register are still used a lot more often than others. r3 for example is both used for the first parameter and for the return value. r1 is the stack pointer, which is also used often. Quite precise statistics are very easy to make in a UNIX environment like Mac OS X. The following shell script dumps register usage statistics within all GUI executables:

```
for i in *; do
    i=$(echo $i | sed -e s/.app//g);
    echo "/Applications/$i.app/Contents/MacOS/$i";
    otool -tv "/Applications/$i.app/Contents/MacOS/$i" >> disass.txt;
done
for i in r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15 r16 \
r17 r18 r19 r20 r21 r22 r23 r24 r25 r26 r27 r28 r29 r30 r31 \
ctr lr bl.0 bcl \\. cr1 cr2 cr3 cr4 cr5 cr6 cr7; \
do (t=$(grep $i[^0-9] disass.txt| wc -l); echo $t - $i); done
t=$(grep cmp disass.txt| grep -v cr[1-7] | wc -l); echo cr0 - $i
```

[2]Theoretically, it is possible to use only one scratch register, by ”push”ing register+20*4 and popping it into 8(%eax), but this is very impractical.

Figure 3.6: PowerPC Register Usage Frequency

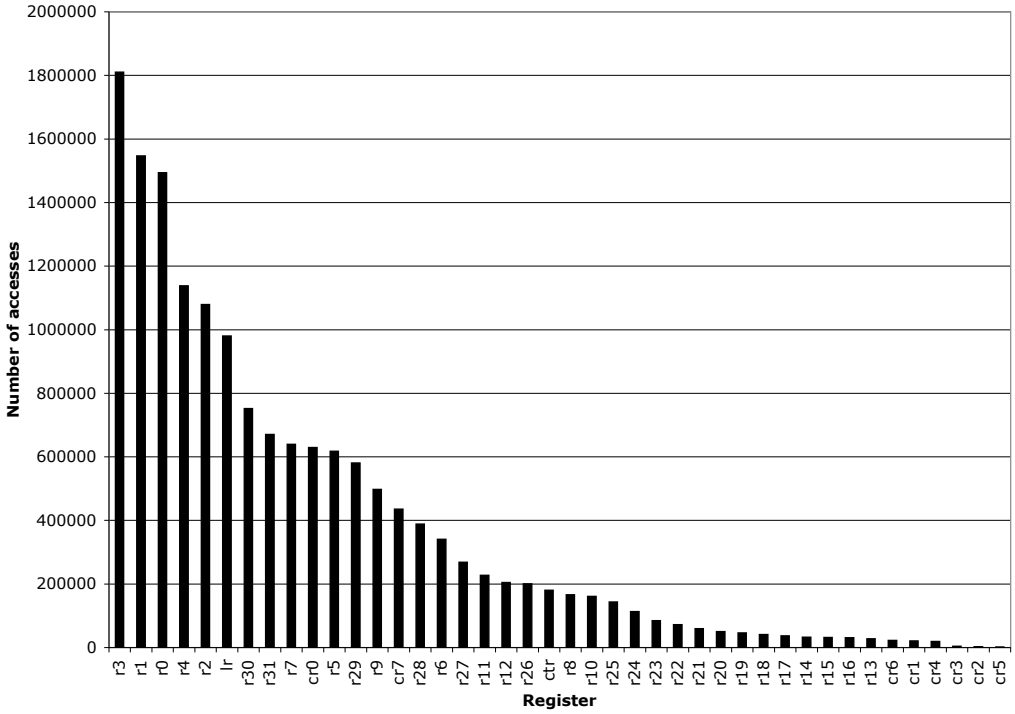


The condition code registers cr1 to cr7 can easily be counted, as they are explicitly mentioned in the disassembly. The use of cr0 is found out by counting all ”cmp” instructions that do not use any other condition register and all arithmetic and logic instructions that modify cr0 (instructions with the ”.” suffix). The real number of cr0 accesses should be about twice as high, as compare/dotted instructions and conditional jumps should always come in pairs.

Since the link register can be used implicitly (”bl” and ”bcl”), it is being searched for using ”lr”, ”bl” and ”bcl”; the results have to be added afterwards. The representation of ctr is not completely fair, because conditional jumps that modify ctr are not counted, but ctr is used rarely anyway. It should also be noted that this script only counts the number of instructions that access a certain register, not the total register accesses - a single instruction can read and write the same register for example. Thus, these statistics are not infinitely precise, but good enough.

Analyzing over 10 million instructions from about 80 Mac OS X GUI applications results in figures 3.6 and 3.7. r0 and r2 are popular, because they are typically used as scratch registers by compilers. r1 is the stack pointer and it is of course heavily used to access the stack. As mentioned earlier, r3 contains both the first parameter of a function and the return value, so it is no wonder that it is the most frequently accessed register. As the registers starting with r3 are used as parameters, these are used more often. The same is true for r31 and the following registers (counting downwards), because these must be preserved by a called function and thus often contain values that the caller will need after the call. The

Figure 3.7: PowerPC Register Usage Frequency (sorted)



most frequent condition registers are cr0 and cr7, the other conditional registers are hardly ever used.
The six most frequently used registers are r0, r1, r2, r3, r4 and lr. About 51% of all register accesses use these registers. This is a very good quota for the static allocation algorithm: Although only 18% of the PowerPC registers can be mapped to i386 registers, about 51% of all accesses will be register mapped. Fortunately, the PowerPC stack pointer r1 is among these six registers, so the i386 ESP register can be used to map r1 without wasting a register. Interestingly, cr0 is only on position 10, but cr0 and cr7 combined would be on position six.

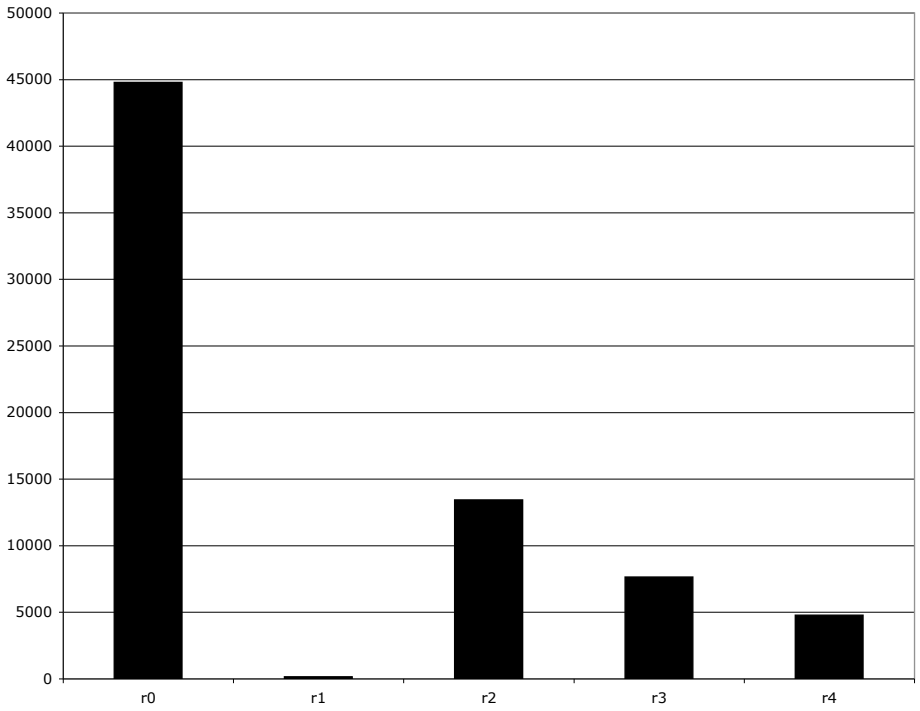
Mapping EAX and EDX are scratch registers. r1 will be mapped to ESP. What about the other registers? Basically, it does not matter. Two of the remaining five i386 registers (EBX and ECX) can be used as 8 bit registers at well, so they are more valuable than ESI, EDI and EBP. For example, if r3 is mapped to EBX, the instruction ”stb r3, -16(r1)” can be translated into:

```
mov -16(%esp), %bl
```

But if r3 is mapped to ESI, it would have to be translated, using a scratch register, into:

```
mov %esi, %eax
mov -16(%esp), %al
```

Figure 3.8: PowerPC Register Usage Frequency (Byte Access)



The script shown earlier can be used to find out what registers are most often used for byte accesses (figure 3.8).

It is not surprising that the stack pointer is virtually never used to load bytes from memory. The scratch registers r0 and r2 are used more often for this purpose, they do about 54% of all byte accesses. So r0 and r2 should be mapped to EBX and ECX.

The remaining three i386 registers are the same, none has an advantage or a disadvantage, so all possible mappings would be basically the same. Though, the produced code is more readable, if the PowerPC link register is mapped to EBP: In typical i386 code, the EBP register is practically never used for calculations and only for addresses, and is therefore similar to the PowerPC link register.

So the resulting mapping looks like this:

EAX	scratch
EBX	r0
ECX	r2
EDX	scratch
ESI	r3
EDI	r4
EBP	lr
ESP	r1

Link Register Emulation An interpreter would emulate the link register just like every other register. It is just another register that can contain a 32 bit value and can be read and written. In practice, it always contains the return address to the caller in the PowerPC address space, although the interpreter need not be aware of this fact.
So recompiled code cannot just do a ”return” sequence as it would be done on the PowerPC (”jmp *lr”), because the address in the emulated link register points to the address of the original PowerPC code that called this function instead of to the translated code. The address would have to be translated first. This could be done by either jumping back to the recompiler that can look up the address in a table, or by inserted recompiled code that can read the value from the recompiler's tables independently.
Both methods are slow, because an address has to be looked up every time a function returns. If the translated code has to be independent of external data structures and fast, the link register has to contain a pointer to the i386 code that resumes execution. Given EBP is the i386 link register, this would lead to the following code that represents a function call (”bl”):

```
    mov $next_address, %ebp
    jmp function
next_address:
```

On a PowerPC, ”bl” writes the address of the next instruction into the link register and jumps to the given address. There is no i386 instruction that does the same, so two instructions are necessary. The code above also writes the address of the next instruction, this time an i386 instruction, into the link register and jumps to the given address. One disadvantage of this method is that the i386 code is no longer relocatable, as it contains explicit absolute references to code addresses within the code[3]. Furthermore, the produced code is quite big (11 bytes), because two 32 bit addresses have to be encoded instead of just one.
A return instruction (”blr”) can be encoded a lot more easily:

```
    jmp *%ebp
```

This i386 instruction jumps to the address that the EBP register points to. This is analog to the original ”blr” behavior, which jumps to the address that the link register points to. This method is not identical and fully compatible to the original behavior, though: The emulated link register contains i386 addresses instead of the original PowerPC addresses. An application could easily copy the link register into a general purpose register and find out that it might not even point into PowerPC code. In user mode, though, it is very unlikely that an application uses the link register for anything other than saving return addresses without ever looking at its contents.
It could be argued that this introduction of a link register to the i386 architecture can actually speed up execution and that using a link register is faster than storing the return address on the stack. Unfortunately, this is not true. i386 CPUs are optimized for call/ret subroutines: They manage return addresses using an internal return stack, so they know the return

[3] If relocatability is required, the sequence ”call nextline” ”nextline:” ”pop %ebp” ”add $11, %ebp” ”jmp function” could be used, with obvious large size and execution time costs.

Table 3.4: PowerPC condition code bits

meaning	PowerPC cr0
$<$	8
$>$	4
=	2

Table 3.5: PowerPC condition code evaluation

bit #	jump if	meaning
1	=0	$\neq$
1	=1	$=$
2	=0	$\leq$
2	=1	$>$
3	=0	$\geq$
3	=1	$<$

address well before the ”ret” instruction reads the address from the conventional stack [37, p. 96]. Calculated jumps do not enjoy these optimizations. If the return address is not known early enough, this will lead to a pipeline stall and thus to a significant delay. The emulation of a link register is not optimal, but required, in order to keep the stack contents compatible.

3.3.3 Condition Code Mapping

As stated earlier, the condition code respectively flags of the PowerPC and the i386 architecture are very incompatible. Although the pairs of instructions that are most frequently used in connection with condition codes/flags (like cmp/bne and cmp/jne) look similar, their internal behaviour is entirely different. It is not possible to just translate the mnemonics, since on the PowerPC, the compare is signed/unsigned-aware, while on the i386, the condition jump cares about signed/unsigned. Furthermore, the PowerPC has eight condition registers, and many instructions can access any of them.
The contents of cr0 (condition register 0) after a ”cmpi rA, SIMM” (signed compare) illustrates the condition code system of the PowerPC, as shown in table 3.4. A conditional jump evaluates a condition register by branching if a certain bit is either set or cleared, as shown in table 3.5.

Table 3.6: i386 flags

meaning	i386 flags
signed $<$	S=1, Z=0
signed $>$	S=0, Z=0
signed $=$	S=0, Z=1
unsigned $<$	C=1, Z=0
unsigned $>$	C=0, Z=0
unsigned $=$	C=0, Z=1

Table 3.7: i386 flags evaluation

instruction	test	meaning
jg	S=0 $\wedge$ Z=0	signed $>$
jl	S=1	signed $<$
jge	S=0	signed $\geq$
jle	Z=1 $\vee$ S=1	signed $\leq$
je	Z=1	$=$
ja	C=0 $\wedge$ Z=0	unsigned $>$
jb	C=1	unsigned $<$
jae	C=0	unsigned $\geq$
jbe	Z=1 $\vee$ C=1	unsigned $\leq$

A compare instruction on an i386 sets the EFLAGS register according to table 3.6. ”S” is the sign flag, ”C” the carry flag and ”Z” the zero flag. The compare instruction does not care whether the operands have been signed or unsigned, so it sets all three flags. The ”jcc” conditional jumps evaluate the EFLAGS register as shown in table 3.7. Depending on whether the conditional jump treats the operands of the compare instruction as signed or unsigned values, the sign or the carry flag is evaluated.
Because the pass 1 of the recompiler translates all instructions independently, the complete information that is returned by the PowerPC compare instruction must be saved so that the conditional jump instruction can access it. On an i386, the information returned by a compare is the result if the values are regarded as unsigned integers, as well as the result if the values are regarded as signed integers. On the PowerPC, this information consists of the actual result, and the information whether the comparison regarded the operands as unsigned or as signed values. When translating from PowerPC to i386, the missing information, that is, whether the compare has been signed or unsigned, must be saved together with the result.
There are several solutions for the representation of the PowerPC condition codes on an i386.

3.3.3.1 Parity Flag

This solution stores the i386 flags, extended by the signed/unsigned information, in the emulated condition register.

```
cmp $5, %esi
lahf
and $%11111011, %ah
mov %ah, flags+0
[...]
mov flags+0, %ah
sahf
jp signed
jbe label1
```

```
jmp label2
signed:
jle label1
label2:
nop
```

”lahf” transfers the i386 flags into the ah register. The parity flag, which is otherwise unused, is used to signal that the compare was unsigned. This value is then stored in a memory location that represents the first PowerPC condition register (cr0). Instructions that use other condition registers than cr0 would be translated accordingly.
When the flags need to be evaluated, the value will be read from memory again and copied into the i386 flags register (”sahf”). If the parity bit is set, it has been a signed compare, and the signed jcc instructions need to be used (label ”signed”).
This solution needs 3 instructions for the compare sequence (not counting the ”cmp”) and 4-5 instructions for the conditional jump. This makes a total of 7.5 instructions - but the cascaded conditional jumps are disastrous for any CPU pipeline.

3.3.3.2 Conversion to Signed Using a Table

It makes more sense to convert the i386 flags at the very beginning into a format that can be evaluated more quickly afterwards.

```
cmp $5, %esi
lahf
[ mov flags_unsigned_to_signed(%ah), %ah ]
mov %ah, flags+0
[...]
mov flags+0, %ah
sahf
jle label1
nop
```

i386 flags stored in memory should always be in a format so that they can be correctly evaluated with the signed jcc instructions. So in case of an unsigned compare, the resulting flags have to be converted, either by copying the C flag (bit #0) into the S flag (bit #7) or by using a 256 bytes table (flags_signed_to_unsigned). In case of a signed compare (assuming signed compares are more frequent than unsigned ones, which is typically the case[4]), no such conversion has to be done. To evaluate the flags, the flags value has to be copied back into the flags register and a signed jcc is enough for the branch.
This solution needs 2 (signed compare) or 3 (unsigned compare) instructions for the comparison and 3 instructions for the conditional jumps. This makes a total of 5.5 instructions. The memory access we need for all unsigned compares ($\leq$50% of all compares) might be a performance problem, though.

[4]This can be proven easily by counting cmpw and cmpwi instructions in a disassembly: 80% of all compares are signed - and instructions with a ”.” suffix are not even counted yet.

3.3.3.3 Conversion to PowerPC Format

```
    xor %eax, %eax
cmp $5, %esi
lahf
shr $eax, 8
mov flags_unsigned_i386_to_ppc(%eax), %ah
mov %ah, flags+0
[...]
test $4, flags+0 // 4: ">"
jz label1
nop
```

This solution converts the i386 flags into PowerPC format, using one of two 256 bytes tables, one for signed and one for unsigned flag conversion. The result is then stored in memory. To evaluate it, there is no need to load it back into the i386 flags register; a ”test” instruction is enough. This instruction tests whether one or more bits are set or cleared. The combination test/jcc is directly analog to the PowerPC branch, which tests a single bit and branches in one instruction.
This solution always needs 5 instructions for the compare, and 2 instructions for the conditional jump. This makes a total of 7 instructions, one of which is a memory read.

3.3.3.4 Intermediate i386 Flags

This is a combination of the last two methods. We store the i386 flags in memory in case of a signed compare, and the signed-converted flags in case of an unsigned compare. The evaluation is then done using the test instruction instead of loading the flags value back into the flags register:

```
cmp $5, %esi
lahf
[ mov flags_unsigned_to_signed(%ah), %ah ]
mov %ah, flags+0
[...]
test $0xc0, flags+0 // Z=1 || S=1: "<="
jnz label1
nop
```

Table 3.8 contains all five cases and the code needed for them. Just like in method 2, the flags conversion is only necessary for the less frequent unsigned compared. This solution needs 2 or 3 instructions for the compare, and two for the conditional jump. This makes a total of 4 or 5 instructions. A memory read is needed in at most 50% of all cases.

3.3.3.5 Memory Access Optimization

In case of an unsigned compare, the i386 flags, which are set correctly for an unsigned conditional jump, must be adjusted so that a signed conditional jump instruction will have

Table 3.8: Testing i386 flags in registers using ”test”

meaning	i386 flags	how to test
$>$	S=0 $\wedge$ Z=0	`test $%11000000; jz`
$<$	S=1	`test $%10000000; jnz`
$=$	Z=1	`test $%01000000; jnz`
$\geq$	S=0	`test $%10000000; jz`
$\leq$	Z=1 $\vee$ S=1	`test $%11000000; jnz`

the same effect. The previous solutions used a 256 bytes conversion table in memory and are therefore slow. A faster sequence of instruction is needed that copies the carry flag (bit #0) into the sign flag (bit #7):

```
and $0x7f, %ah
test $1, %ah
jz label1
or $0x80, %ah
label1:
```

This method tests whether bit 0 is set, and if it is, bit 7 is set. But conditional jumps are very expensive.

```
mov %ah, %al
shl $7, %al
and $0x7f, %ah
or %al, %ah
```

This method shifts a copy of the value left by 7 and combines the results. This is faster, as it does not need a conditional jump, and consists of just as many instruction.
Melissa Mears [66] developed the following idea:

```
lea (%eax,%eax), %eax
rcr $1, %ah
```

The lea shifts EAX left by one without modifying the carry bit — which had been set by the cmp — and the rcr shifts AH right again, placing the carry flag in bit 7 and restoring the position of the zero flag. The other 24 bits of EAX are unimportant here, allowing us to destroy them with the ”lea”. This is most probably the optimal sequence of instructions to do the job.

3.3.3.6 The Final Code

So in case of a signed compare, the generated code looks like this:

```
cmp $5, %esi
lahf
mov %ah, flags+0
[...]
test $0xc0, flags+0
jnz label1
nop
```

Unsigned compares are longer by 2 instructions:

```
cmp $5, %esi
lahf
lea (%eax,%eax), %eax
rcr $1, %ah
mov %ah, flags+0
[...]
test $0xc0, flags+0
jnz label1
nop
```

So a 2 instructions PowerPC compare/branch sequence is expanded into 4 resp. 5 i386 instructions with two memory accesses. The memory accesses could be avoided if ah is unchanged between the compare and the conditional jump, which is hard to detect when translating instructions independently. Alternatively, the most important condition register CR0 could be statically mapped to an i386 register:

```
cmp $5, %esi
lahf
mov %ah, %bl
[...]
test $0xc0, bl
jnz label1
nop
```

This method could even map two condition registers to an i386 register if one the the first four i386 register is chosen, as these can be divided (BL and BH in this example; the upper 16 bits cannot be accessed separately). Though, as the statistics show that compilers seem to use cr0 and cr7 about equally, so none of them is used frequently enough to be more important than the most frequent GPRs. As a consequence, it is more important to map a general purpose register instead of parts of CR to an i386 register.

3.3.3.7 Compatibility Issues

All these methods assume that a condition register can only contain the values of 2, 4 or 8, i.e. only one of the three bits is set. All compare instructions as well as instructions with a "." suffix in the mnemonic will always leave the condition register in a state that complies

with this assumption. But there are other instructions that make direct access to the bits of the condition registers possible. Using these instructions, it is possible to set more than one bit, or even clear all bits. In case all three bits are set, this would have the meaning that the result of the last comparison shows that the first operand was above, below and equal to the second operand. This makes little sense, but it is possible. One possible use might be to store an arbitrary 32 bit value in the 8 condition registers.
In practice, this is most probably extremely rare. Nevertheless, it would be no problem to implement the perfectly compatible method (3.3.3.3) in parallel and allow switching between the two methods either at compile time or at runtime.

3.3.4 Endianness

A PowerPC is typically big endian, and i386 CPUs can only do little endian, so the endianness problem has to be addressed somehow. In some cases, this problem is easy to solve: If the interpreter of the instruction decoder read from memory, the endianness can be easily converted by a function that abstracts memory accesses. Accesses done by the recompiled code can be handled the same, but a function call for every memory access is certainly a performance problem. There are three basic ways to handle it the endianness problem efficiently:

3.3.4.1 Do Nothing

The simplest solution is certainly to do nothing. This is possible if there is no endianness problem, i.e. for example, if both CPUs have the same byte order (or the target CPU can be switched to the byte order of the source CPU), if the source CPU does not have a byte order at all, or if the code in the source language is written in a way that no endianness problems can ever happen (only aligned machine words accesses). Otherwise, this solution is not compatible and code execution will likely be faulty.
But even if the source language is written to avoid endianness problems, the code segment and the (predefined) data segment must be byte swapped when loading. This method is mainly useful during development, to keep the translated code easy to read and to debug.

3.3.4.2 Byte Swap

The most common method to solve the endianness problem is to byte swap every value before it is written to or after it is read from memory. Beginning with the i486 (introduced in 1989), the i386 architecture provides the ”bswap” instruction. It take a single register as an argument and converts it between the little and big endian conversions:

```
before 12 34 56 78
after  78 56 34 12
```

When reading a 32 bit value from memory, it must be byte swapped like this:

```
mov 0x12345678, %eax
bswap %eax
```

When writing a register to memory, it has to be byte swapped before, and, because the ”bswap” instruction overwrites the original value, it has to be byte swapped again afterwards to restore the original value:

```
bswap %eax
mov %eax, 0x12345678
bswap %eax
```

Alternatively, the register can be copied into a temporary register and swapped, so it does not have to be restored afterwards:

```
mov %eax, %edx
bswap %edx
mov %edx, 0x12345678
```

16 bit values can easily be byte swapped using the ”xchg” instruction, which exchanges the two bytes in the lower 16 bits of a register. When reading from memory, it looks like this:

```
mov 0x12345678, %ax
xchg %al, %ah
```

When writing to memory, as before, the operation has to be restored afterwards:

```
xchg %al, %ah
mov %ax, 0x12345678
xchg %al, %ah
```

As before, it is possible to use a temporary register to achieve the same effect:

```
mov %ax, %dx
xchg %dl, %dh
mov %dx, 0x12345678
```

Since only the lower four i386 register (EAX, EBX, ECX, EDX) support the splitting into 8 bit registers, only the second option is possible when storing one of the other registers into memory.
Some Read-Modify-Write operations (like memory increment), as they are supported by the i386, are not possible any more if memory is in a different endianness then the CPU. Instead of writing

```
incl 0x12345678
```

the following sequence would have to be used:

```
mov 0x12345678, %eax
bswap %eax
inc %eax
bswap %eax
mov %eax, 0x12345678
```

Fortunately, this is never necessary. Being a RISC architecture, the PowerPC has no Read-Modify-Write instructions that operate on memory. PowerPC registers can be mapped to i386 memory locations, though, so the i386 might still have to do Read-Modify-Write on memory. But in this case, endianness is no problem: Neither register mapped nor memory mapped registers have to be converted, as registers are internal to the CPU and are not even endianness-aware.
The ”bswap” method to cope with the endianness problem might not produce very beautiful code. Every memory access is accompanied by additional ”bswap” instructions, which also slow down the code to some extent. The ”dynarec” group [25] measured a performance penalty of about 5% in a dynamic recompiler that translates M68K to i386 code [65]. But loads and stores are a lot rarer on RISC CPUs than they are on CISC systems, as they are typically only necessary when modifying data structures in memory, so most of the translated code is no different than code that ignores the endianness problem.

3.3.4.3 Swapped Memory

Alternatively, all data can be stored in native byte order, i.e. in reverse byte order, from the view of the source CPU. The loader must interpret the code and data segment as an array of 32 bit values and byte-swap all of them. Afterwards, all 32 bit memory accesses are left unchanged, which is a great performance advantage, and all 8 and 16 bit accesses (which are likely to be less frequent) must be taken care of. Unaligned 32 bit accesses (also typically less frequent) must be trapped by the CPU logic.

32 Bit Accesses As it doesn’t matter in what format data is stored in words in memory if they are only read and written word-wise, endianness can be just ignored when doing 32 bit accesses:

```
li r3, 0x1000 |    mov $0x1000, %esi
li r0, 23     |    mov $23, %eax
stw r0, 0(r3) |    mov %eax, (%esi)
lwz r1, 0(r3) |    mov (%esi), %ebx
```

This code stores the value of 23 in memory and reads it again. Both the PowerPC and the i386 version read back the same value again, although it is stored differently in memory.

8 Bit Accesses But this does not work when mixing 32 and 8 bit accesses:

```
li r3, 0x1000 |    mov $0x1000, %esi
li r0, 23     |    mov $23, %eax
stw r0, 0(r3) |    mov %eax, (%esi)
lbz r1, 0(r3) |    mov (%esi), %bl
```

This time, a 32 bit value is stored in memory, and an 8 bit value is read from the same location. In big endian, this is the uppermost 8 bits, and in little endian, this is the lowermost 8 bits. So the PowerPC would read ”0” and the i386 would read ”23”. So all 8 and 16 bit accesses and unaligned accesses must be converted. The following code corrects this:

Table 3.9: Byte ordering differences

CPU	PowerPC	i386
byte number	0123	0123
order	HhLl	lLhH
*256^	3210	0123

Table 3.10: Endianness offset differences

PowerPC	i386
0	3
1	2
2	1
3	0

```
li r3, 0x1000 |    mov $0x1000, %esi
li r0, 23     |    mov $23, %eax
stw r0, 0(r3) |    mov %eax, (%esi)
lbz r1, 0(r3) |    mov 3(%esi), %bl
```

The uppermost 8 bits of the 32 bit value are stored at the address + 3, so the i386 has to read from a different address. Table 3.9 summarizes the differences of the byte offsets when addressing bytes within an int.
If the i386 stores all 32 bit values in its native format, byte accesses have to be adjusted to that the same byte is read that would have been read by a PowerPC. The last two bits of every address have to be taken into account: A value of 0 becomes 3, 1 becomes 2 and so on, as illustrated in tables 3.10 and 3.11.
So the conversion is basically an XOR with 0b11. When doing byte accesses, all addresses have to be XORed with 3. This can be done at compile time if the address is well-known (modify the constant address) or at runtime (modify the address register, do the access, then modify it back, or copy address register into a scratch register and modify it). The example above would be translated into the following i386 code:

```
li r3, 0x1000 |    mov $0x1000, %esi
li r0, 23     |    mov $23, %eax
stw r0, 0(r3) |    mov %eax, (%esi)
lbz r1, 0(r3) |    xor $3, %esi
              |    mov (%esi), %bl
```

Table 3.11: Endianness offset differences (binary)

PowerPC	i386
00	11
01	10
10	01
11	00

Table 3.12: Unaligned access conversion

a & 3	result	
0	swap(halfword(a+2))	(aligned)
1	swap(halfword(a))	(unaligned)
2	swap(halfword(a-2))	(aligned)
3	byte(a-3) << 8 + byte(a+4)	(unaligned over 4 byte boundary)

```
                |    xor $3, %esi
```

The recompiler cannot just XOR the displacement with 3, as done in the manual translation before, as in general, the compiler cannot find out whether r3/ESI is aligned in this case - adding 3 to a value is only the same as XORing it with 3 if the value is divisible by 4.

16 Bit Accesses 16 bit accesses look the same, if they are aligned:

```
li r3, 0x1000 |    mov $0x1000, %esi
li r0, 23     |    mov $23, %eax
stw r0, 0(r3) |    mov %eax, (%esi)
lhz r1, 0(r3) |    mov 2(%esi), %bx
              |    xchg %bh, %bl
```

If the address is read from a 32 bit boundary, the resulting 16 bits have to be read from the memory location + 2, and the two bytes have to be swapped afterwards. Again, this code is only possible, because it is known that r3/ESI is aligned.
In general, it is more complicated, as tabe 3.12 illustrates. Again, if the address is a constant, the recompiler can emit code that adjusts the address and swaps the data. But if the address is a variable, the test has to be done at runtime.

Unaligned 32 Bit Accesses Unfortunately, 32 bit accesses can only be ignored if they are aligned. The following code demonstrates this:

```
li r3, 0x1000 |    mov $0x1000, %esi
li r0, 23     |    mov $23, %eax
stw r0, 0(r3) |    mov %eax, (%esi)
li r0, 17     |    mov $17, %eax
stw r0, 4(r3) |    mov %eax, 4(%esi)
lwz r1, 1(r3) |    mov 1(%esi), %eax
```

This code writes the values of 23 and 17 into two adjacent 32 bit fields into memory and reads from the address of the first value + 1. This is the order of the byes in memory on a PowerPC:

```
 XX XX XX XX
| 0| 0| 0|23| 0| 0| 0|17|
    XX XX XX XX
```

Reading from the start address + 1 reads the bytes 0, 0, 23 and 0, resulting in a value of 23*256. On the i386, memory looks like this:

```
 XX XX XX XX
|23| 0| 0| 0|17| 0| 0| 0|
    XX XX XX XX
```

So it reads 0, 0, 0, 17, resulting in 17*256ˆ3.
Unaligned accesses could be handled by inserting additional code on every 32 bit access that checks whether the access is aligned or unaligned, but this would undo the advantage of not needing extra code for 32 bit accesses. Instead, these accesses can be trapped by the CPU. The i386 can throw an exception on a misaligned access. The following code shows this:

```
.align 4
.globl main
main:
pushfl
label1:
popl %eax
or $0x00040000, %eax
pushl %eax
popfl
movl (label1), %eax // label1 is unaligned
ret
```

This code sets a flag in the i386 EFLAGS register to enable misalignment exceptions. While Windows does not hand down these exceptions to the application, both Linux and FreeBSD do, as SIGBUS signals. The example terminates with a ”bus error” if the ”or” is enabled, and terminates cleanly if the ”or” is disabled. So it should be possible to trap misaligned accesses, handle them, and return to the original code.

3.3.4.4 Conclusion

During development of the recompiler, as long as the possibility of endianness problems can be excluded, ”do nothing” is possible - but the code segment still has to be byte swapped either already by the loader, or by the instruction decoder. This method produces the most readable code that is well-suited to debug the rest of the system.
While ”bswap” is easier to implement and more portable between host operating sytems, the ”swap memory” method most probably achieves a higher performance. The performance penalty of the ”swap memory” method compared to the ”do nothing” method is hard to estimate, but probably not zero. The difference in overall performance of an application, if the ”swap memory” and the ”bswap” method are compared, will probably not be worth the effort to implement the more complicated method. As the endianness strategy is no fundamental design decision that has effects on the overall design, ”bswap” can be implemented now, and ”swap memory” can be added as a compile time option later.

3.3.5 Instruction Recompiler

The instruction recompiler translates every instruction independently, that is, it is given a single instruction, it emits the resulting target code and returns to the translation loop afterwards. Instructions are translated in four steps, similarly to a pipeline in a CPU. These four steps are:

1. The dispatcher detects the type of instruction and jumps to the appropriate part of the decoder.
2. The decoder extracts the operands from the instruction and translates the instruction into one or more operations.
3. The i386 converter maps source registers to target registers and converts one operation into a sequence of i386 operations.
4. The instruction encoder emits the optimal i386 machine code encoding of an i386 operation.

The conversion of PowerPC code to i386 code in two steps between 2 and 4 is similar to a conversion with intermediate code. In this case, there is no explicit intermediate code, but all intermediate information is represented by the direct data flow between the steps.

3.3.5.1 Dispatcher

Every instruction of PowerPC binary code has a 6 bit ”primary opcode” field, and depending on the value of this field, it can optionally have another 9 or 10 bit ”extended opcode” field. The function of the dispatcher is to decode the opcode field or fields and execute the piece of code of the decoder that is responsible for the instruction that has been detected. All information that has to be passed to the decoder is the 32 bit instruction code and the current program counter.
For example, the PowerPC instruction ”or” has a primary opcode of 31 and an extended opcode of 444. The dispatcher extracts the opcode fields, detects that it is the ”or” instruction and executes the ”or” handler of the decoder.

3.3.5.2 Decoder

The function of the decoder is to convert the instruction into a more general form and hand the information down to the i386 converter. It first extracts all operands, that is registers, constants and flags from the instruction code. Then it checks whether the instruction has a special combination of operands that have a special (simplified) meaning. Depending on the result of this check, the instruction is decomposed into several operations, if applicable, and these operations are then passed to the i386 converter. If the recompiler has to be very fast, for example in pass 1, the check for simplified combinations can be omitted, which will produce code of lower quality.
The instruction code of ”or” for example has three register operand fields, rA, rS and rB, and one flag field ”Rc”, which states whether the the condition register 0 is supposed to be

altered by the result. After these four fields have been extracted, it will be tested for two special cases: If all three register numbers are the same, this corresponds to the instruction ”or rX, rX, rX”, which is a no-operation instruction. But in case the Rc flag is set (”or. rX, rX, rX), the instruction effectively tests whether the value of register rX is positive, negative or zero and sets the condition register 0 accordingly. So the decoder has to check whether Rc is set and either pass nothing (Rc=0) or the command ”set cr0 according to value of rX” to the i386 converter.
If the two source registers fields are the same (”or rX, rY, rY”), the instruction is effectively a ”register move” (”mr”) instruction. The information ”move register rY to rX” is passed to the i386 converter. If Rc is set, ”set cr0 according to rY” is passed in addition.
If all three register numbers are different, it is a conventional or instruction, which will be passed to the i386 converter. Again, if Rc is set, another command, ”set cr0 according to result” will be sent.

3.3.5.3 i386 Converter

The i386 converter has to convert an operation in general form into one or more i386 instructions. In addition, it maps source registers to i386 registers. If static register allocation is used, then register mapping is as easy as looking up the source register in a table, to get either an i386 register index (0-7) or an address that points to the memory location that emulates the source register.
Depending on whether the operands are register or memory mapped, the operation has to be converted into different i386 sequences. For n operands there are 2^n independent parts of the i386 converter. Every part sends one or more i386 instructions, such as loads, stores and arithmetic instructions to the instruction encoder. Endianness conversion is done by the i386 converter as well: In case the input operation is a memory access, additional instructions are added to compensate the endianness difference.
If, for example, the i386 converter gets passed an ”or” operation and three source register numbers, the i386 converter first looks up the source registers in the register mapping table. Given the two input registers of the ”or” operation are, for example, both register mapped, and the output register is memory mapped, the i386 converter branches into one of its eight (2^3) parts, that is, into the memory-register-register part. This part now knows the exact i386 sequence that is needed: In the first instruction, the first input operand has to be copied into a temporary register, in the second instruction the second input operand has to be ORed to the temporary register, and in the third step, the temporary register has to be written into memory. In i386 assembly, this will look like this:

```
mov %esi, %eax
or %edi, %eax
mov %eax, register+8*4
```

ESI and EDI are the i386 registers that represent the input operands, and r8 is the output operand register, which is not register mapped. The i386 converter has the sequence for this combination of source registers (memory-register-register) hardcoded, and the actual i386 register numbers and memory locations have been read from the register mapping table. The instruction encoder will be called three times with one instruction each time.

3.3.5.4 Instruction Encoder

The instruction encoder is supposed to find the optimal encoding for an i386 instruction and emits the i386 machine code. In order to optimize the encoding, special encodings have to be found (some instructions have a redundant shorter encoding if EAX, AX or AL is involved) and redundant, but optimized i386 instructions should be used when appropriate (for example, ”dec” instead of ”sub $1”). As these optimizations might not amortize, especially in pass one, they are optional.
Following up the example, the i386 converter passes a register store to the instruction encoder. The register is EAX and the the address is 0x12345678. The instruction decoder detects that the register is EAX and thus emits the optimized encoding

```
a3 78 56 34 12          mov    %eax,0x12345678
```

instead of

```
89 05 78 56 34 12       mov    %eax,0x12345678
```

The byte sequence of 0xa3, 0x78, 0x56, 0x34, 0x12 will be written into the i386 code buffer.

3.3.5.5 Speed Considerations

This whole concept sounds terribly slow. But it is not. There is no intermediate code that has to be encoded and decoded again. There are no redundant decisions about information that has already been known earlier. All information is directly passed from the dispatcher down to the instruction encoder. This concept just makes heavy use of code reuse by putting a lot of functionality into small functions.
The dispatcher extracts the opcode and jumps directly to the according handler in the decoder. This handler extracts the operands, (optionally) decides whether this combination has a special meaning, and for every operation the PowerPC instruction carries out, the according handler in the i386 converter is called, which converts the source registers, and decides what i386 code to produce. For every i386 instruction the according handler of the instruction encoder is called.
This shows that data flow is very direct. No information ever gets written into memory and read later. Modern compilers can eliminate all function calls to increase the speed further.

3.3.6 Basic Block Logic

The function of the basic block logic is to decide how much code to recompile and what blocks to link. In addition, this takes care of basic block caching.

3.3.6.1 Basic Blocks

A dynamic recompiler always translates code on demand. Only code that is about to be executed will be recompiled. It is a major design decision how much code to translate at

a time. All code following the code flow through jumps and calls, up to the next conditional jump will definitely be executed. But it only makes sense to translate consecutive instructions in the code, because otherwise it would be difficult to reuse blocks.
It is also possible to translate more consecutive instructions than are guaranteed to be executed. This way, the recompiled code will have to jump back to the recompiler less often, but some code will be unnecessarily translated. And the larger the blocks are, the higher is the probability that a new jump target will be found in the block, so the block has to be split and retranslated.
In this project, basic blocks are very similar to the classic basic block. They start at any point where concrete execution has to begin. They are ended by the first control flow instruction. There is no analysis before that finds out possible jump targets within the basic block, so a basic block is not necessarily an atomic unit of execution.

3.3.6.2 Basic Block Cache

All recompiled code is of course cached, since the speed of recompilation comes from the repeated execution of the same blocks of code. As said before, basic blocks in this system are not atomic. When a block is about to be executed, no checks will be made of code from the outside might jump into the middle of the basic block. Instead, a part of the basic block will be translated as a new block if a jump target is found within an existing block. The following example is supposed to illustrate this:

```
1       li r0, 100
2       mtspr r0, ctr
3       li r1, 0
4 loop: addi r1, r1, 1
5       bdnz loop
```

At first, this complete sequence is detected as a single basic block. As soon as the last instruction is executed, a new basic block is found, beginning at the label ”loop”. Instead of splitting the original block into two parts, the complete block is kept and the second part is translated again. This leads to two translated basic blocks, one from line 1 to 5, and one from line 4 to 5, so there is more translated code, which may be bad for the code cache of the CPU. On the other hand, this is less work for the basic block cache than splitting the block, and the resulting code may even be faster, because in the case of splitting, an additional jump would be necessary between the blocks (between instructions 3 and 4), so this method is not just easy to implement, but may even produce slightly faster code.

3.3.6.3 Control Flow Instructions

Every basic block ends with an instruction of control flow. In general, the target address or addresses of this instruction are not known, so the recompiler cannot translate the instruction. Therefore basic blocks are always translated without the control flow instruction at the end. Instead, code is inserted that jumps back to the recompiler:

```
mov $0x1eec, %eax
jmp recompiler_continue
```

This sequence loads the address of the next PowerPC instruction into the first scratch register and returns to the recompiler. The recompiler's main program will then hand the following instruction (at address 0x1eec in this case) to the interpreter and continue execution at the effective target address of the control flow instruction.
There is one exception to this behavior: As an optimization, blocks that end with a return ("blr") can include the translation of this instruction. In i386 code, this will look like this:

```
jmp *%ebp
```

This i386 instruction jumps to the address that EBP, which is the link register, is pointing to. There is one problem to this, though: It must be made sure that the EBP register always points to i386 code. When a function is called, the correct address of the i386 instruction following the function call must be placed into the EBP register. This address can only be available if the block following the function call instruction is already translated.
One solution would be to allow function call instructions in basic blocks, that is, to end basic blocks with any control flow instruction other than "bl", but this has the problem that at the time the i386 equivalent of the function call instruction has to be emitted, the first block of the function that is supposed to be called might not be translated yet. This would make the recompiler more complicated.
A simpler solution is to end basic blocks if a "bl" is encountered and just always translate the following block immediately, as a separate block. In the first run, the function call instruction will be interpreted, but the second time, the three blocks will be linked.

3.3.6.4 Basic Block Linking

The "context switch" between the recompiled code and the recompiler, which has to be done before and after the execution of every basic block, is very expensive. A typical basic block consists of less than 10 instructions, and its recompiled version should execute, even pessimistically, still in less than 20 clock cycles. A context switch includes at least exchanging the complete register set, which consists of 16 memory accesses and therefore roughly 40 clock cycles. For every block, there are two context switches, plus the lookup of the following block, all in all well over 100 cycles. These context switches really have to be removed.
A basic block can have two successive blocks (conditional jump, function call), one successive block (jump) or none (blr, but this case has already been handled). If all successive blocks are translated as well, there is no need to go through the recompiler on exit of the block. Instead, the blocks should be linked.
There are enough free bytes at the end of every block to replace the jump into the recompiler with the (possibly longer) code that connects the block with its successor(s). In case the block ends with a jump to another block, all that has to be done is overwrite the jump back to the recompiler with a jump to the next block:

```
jmp block_next
```

A conditional jump will be translated into the according sequence, as discussed earlier, plus a jump to the non-taken block:

```
test $0xc0, flags+0
jnz block_taken
    jmp block_not_taken
```

In case of a ”bl”, the code has to load the address of the block that succeeds the call into EBP (the link register) and jump to the first block of the function:

```
    mov %ebp, block_next
    jmp block_sub
```

Every time a block is supposed to be executed or just has been executed (the block might already have been jumped to by another block, so it might never be started separately), the basic block logic must test whether the block can be linked to its successor or sucessors. So if a certain sequence of code is executed some times, all code around it should be translated, and all code should be linked. Unfortunately, there is a special case. In the following example, the basic block on the bottom will not be linked to the following block during all iterations of the loop:

```
    li r0, 100
    mtspr r0, ctr
    li r1, 0
loop:
    addi r1, r1, 1
    bdnz loop
    [...]
```

If this code is executed for the first time, the block after the loop has never been executed, so it is not yet translated. So one of the targets of the ”bdnz” instruction is still not available in i386 code even after many iterations of the loop. Therefore ”bdnz” cannot be translated into native code, and it has to be interpreted in every iteration, accompanied by two mode switches per iteration. One solution would be to do at least half of the linking already, as one target is already known. But there is a simpler solution: Every time a block has been translated that ends with a branch to a lower address, the succeeding block must be translated as well. The next time the first block is executed, it will be linked with its succeeding blocks. Branches to lower addresses are almost certainly loops.
The possible disadvantage is that there is a chance that the block after the loop may never be reached, because some statement within the loop branches to a different location to exit the loop, so the block might have been translated unnecessarily. But as so often, simplicity should be in favor over tiny optimizations.

3.3.6.5 Interpreter Fallback

As stated earlier, there is no interpreter pass in this system. However, there must be some interpreter code: All control flow instructions of blocks that have not been linked yet are interpreted. So the interpreter only needs to support control flow instructions.

3.3.7 Environment

The recompiler engine needs to be supported by some additional components that provide the emulation environment, such as the loader, the memory management, the disassembler and the interface to the recompiled code (execution/context switches).

3.3.7.1 Loader

The recompiler requires a loader in order to get the executable file into memory. For the recompiler, it does not matter what format the executable file has, so the loader can be replaced with another one that supports a different executable format at any time. All it has to do is move code (and data) into memory and pass the information where it has been loaded, to the actual recompiler.

3.3.7.2 Memory Environment

The loader does not need to put the code from the file at the same position in memory where it would be located if the application was run natively. Not all host environments might make it possible to use arbitrary memory locations to store data, so the system is more independent of the host environment if the code segment is loaded to memory that has been conventionally allocated. If a host operating system supports reserving memory at defined locations, this behavior can still be optimized.

Memory for the target code will also just be allocated; in this case, the address does not matter at all. As the system will try to execute code in memory that has been allocated for data, a problem might arise with newer CPUs and operating systems. Intel, AMD and Transmeta either announced or already introduced i386 CPUs that support the ”NX” (”not executable”) feature, which can disable the possibility to execute code in protected regions, like the stack and the heap, if the operating system supports it. As soon as these CPUs are common, additional code might have to be introduced to fix this.

Although some PowerPC registers are kept in i386 registers, there must still be an array in memory that can contain all PowerPC registers. Memory mapped registers will always be stored in this array, but there also needs to be space for register mapped registers: Between basic blocks, when the system finds the next basic block to execute, all i386 registers have to be written back to memory and registers to support the recompiler environment must be loaded (context switch), so all PowerPC registers that have been mapped to i386 registers will be saved in this array.

The link register also has a representation in memory for this purpose. As shown earlier, it will always point to i386 code. Additionally, the PowerPC program counter will always be stored in memory, although it is only updated every time unknown code is encountered.

Stack space will just be allocated on the heap and the stack pointer will be set pointing to the top of that memory area. User mode applications should not care about the actual location of the stack. Operating systems usually also do not guarantee a certain location of the stack, but most environments place it at some typical location.

3.3.7.3 Disassembler

A PowerPC disassembler is very useful for debugging purposes. If the recompiler is in verbose mode, it can print every instruction on the screen when it is working on it, and print additional information about translation, to make it easier for debugging to find out what source instruction has produced what target instruction(s). So the disassembler should only print one instruction at a time, so that it can be called by the recompiler loop.
No i386 disassembler is necessary for this project. The GCC toolchain comes with ”objdump” that can disassemble even raw files into i386 AT&T syntax. The easiest way to disassemble the output is certainly to make the recompiler write the output code into a file and call ”objdump”.

3.3.7.4 Execution

As described earlier, in order to execute a basic block, all register mapped registers have to be loaded from the array in memory. But the i386 registers cannot just be overwritten, instead they must be saved.
So executing a basic block looks like this:

- save all (non-volatile) i386 registers
- load register mapped PowerPC register into i386 registers
- jump to the recompiled basic block

Recompiled basic blocks end with a jump back into the recompiler. At this point, the following has to be done:

- save i386 registers into the PowerPC register array in memory
- restore the original i386 registers

While the recompiled code is executing, the recompiler can have no influence on execution, as it is completely disabled. The recompiled code must actively return to the recompiler. So if the recompiled code is in an infinite loop, the recompiler has no way to interrupt it, because the recompiled code has effectively taken over the personality of the process. Fortunately, this is no problem: If the program was natively executed, it would be in the same infinite loop, unless it is caused by flawed recompilation, in which case the system should still be in the process of being debugged.

3.3.8 Pass 2 Design

If code that has been translated by the pass 1 recompiler has been run very often, it makes sense to optimize the code in order to increase its performance. The code produced by pass 1 has several deficiencies that could be eliminated by a better pass 2 recompiler:

- 50% of all source register accesses will be translated into memory accesses

- the condition registers are always emulated in memory
- emulating a link register on the i386 is inefficient
- the overall quality of code might be improved

These problems are now to be addressed and solutions are to be provided. As some of these optimizations assume that the code complies with certain conventions (like the stack convention), they will most probably only work with compiled code.

3.3.8.1 Register Mapping Problem

The statistics made earlier showed that about 50% of all register accesses on the PowerPC are done using the most frequent six registers, which are mapped to i386 registers by the pass 1 recompiler. So 50% of all source register accesses will result in memory accesses on the i386. This percentage does not sound bad, but it does if expressed differently: On average, a PowerPC instruction has two register operands. Every register has a probability of 50% to be register mapped. Consequently, three quarters of all PowerPC instructions will be translated into i386 code that has at least one memory access.
So there is room for optimization:

- If the register usage statistics of a function are very different from global statistics, another set of registers could be register mapped for this function.
- If a function uses *n* registers, it does not necessarily mean that it really needs *n* registers. Compilers typically use more registers than necessary in order to keep register dependencies in the CPU low.

Dynamic register allocation can provide a better register mapping.

3.3.8.2 Condition Code Optimization

The pass 1 method uses i386 flags to emulate PowerPC condition codes without converting them. Nevertheless, this data is then stored in memory and will be read again for the conditional jump instruction. This means two memory accesses for every comparison/conditional jump combination. It is indeed necessary to save the i386 flags somewhere, because they could be overwritten by instructions between the compare and the jump:

```
01     addic.  r3,r3,0xffff |     dec %esi
02     li      r9,0x0       |     xor %ebx, %ebx
03     li      r0,0x1       |     mov $1, %edi
04     beq     label3       |     jz label3
```

This example is supposed to branch if r4 (ESI) has reached the value of zero. The PowerPC behaves like this, but in the i386 code, the flags are overwritten between the ”dec” and the ”jz” by the ”xor” instruction. Modifying the condition codes is optional for every PowerPC instruction, but this is not the case on the i386. In the example, the ”xor %ebx, %ebx”

could be replaced by ”mov $0, %ebx”, which does not alter the flags, but there is no general solution to this problem - there is no way to carry out an or instruction without changing the flags[5].
So the i386 flags have to be saved somewhere - but they can be saved in a register instead of memory. This will certainly only be reasonable if none of the six most frequent GPR is be used more frequently than the condition register, so the condition code will not push out an important GPR. Dynamic register allocation can treat the condition register as an ordinary register and assign an i386 register to it, if used frequently.

3.3.8.3 Link Register Inefficiency

Pass 1 mapped the PowerPC link register to an i386 register quite cleanly. Unfortunately, the i386 is optimized for call/ret style functions and performance is not optimal when emulating a link register. Converting PowerPC style function calls into i386 style has another advantage: The sequence that saves the link register on the stack in case the function calls another function can be omitted on the i386. The PowerPC sequence that saves the link register looks like this:

```
mfspr r0,lr         ; get link register
stw r0,0x8(r1)      ; save link register
```

Restoring the link register looks like this:

```
lwz r0,0x8(r1)      ; get link register
mtspr lr,r0         ; restore link register
```

If the i386 uses call and ret, these sequences can be completely omitted. So in pass 2, bl/blr is supposed to be translated into call/ret, and the impact on the stack is supposed to be handled. In addition, the obsolete sequences have to be detected during translation so that no code will be produced.

3.3.8.4 Intermediate Code

Pass 1 recompilation had to be very fast, so there is no other way than translating all PowerPC instructions individually. For pass 2 it should be considered whether the use of intermediate code might be able to improve the quality of the target code.
All modern high level language recompilers use intermediate code. The source language is translated into intermediate code, which in turn is translated into target code. The intermediate code is an abstract language that is independent both of the source and the target language. It is typically more compact and more powerful on an instruction basis, and easy to decode and edit. The use of intermediate code certainly slows down translation significantly, but there are some very interesting advantages:

1. **Regrouping**: Intermediate code splits instructions into smaller instructions, which might be combined in a different combination.

[5] Saving the flags on the stack and restoring them afterwards is certainly not an option.

2. **Optimization**: It is easy to optimize on intermediate code, also in multiple passes, for simplification, using data flow analysis.

3. **Retargetability**: If intermediate code exists, the front end and the back end of the compiler are less dependent on each other and can therefore be replaced more easily.

4. **Metadata**: Intermediate code can extend the source code with metadata, such as liveness information.

These general advantages are now to be inspected in the context of this recompiler, in order to evaluate if they also conform with the aims of this project.
Intermediate code is a representation whose properties are between the properties of the source and the target language. The more the source and the target language are alike the less sense it makes to have an intermediate language. The ARM to i386 recompiler ARMphetamine [21] for example make use of intermediate code, because despite being a RISC CPU, the ARM has very complex instructions, for example, any instruction can include a shift and conditional execution. ARMphetamine decomposes ARM instructions and encodes i386 instructions by matching patterns of the intermediate code [67].
The PowerPC is unlike the ARM CPU though, and more similar to the i386 than the ARM is. PowerPC instructions can rarely be decomposed into anything else than the actual operation and the update of the condition register. Most PowerPC instructions can be directly translated into one i386 instruction, or, if the PowerPC instruction has three different registers as operands, into two i386 instructions. Also, there are few i386 instructions that could combine PowerPC instructions. The following example is an implementation of the iterative calculation of the Fibonacci number sequence. On the left, there is the original implementation in PowerPC assembly, and on the right, there is the optimal translation of the PowerPC code into i386 code:

```
00 fib:    cmpwi  r3,0x1       |fib:    cmp $i, %esi
01         or     r0,r3,r3     |        mov %esi, %edi
02         ble    label1       |        jle label1
03         addic. r3,r3,0xffff |        dec %esi
04         li     r9,0x0       |        mov $0, %ebx
05         li     r0,0x1       |        mov $1, %edi
06         beq    label3       |        jz label3
07         mtspr  ctr,r3       |        mov %esi, %ecx
08 label2: add    r2,r9,r0     |label2: lea (%ebx,%edi), %esi
09         or     r9,r0,r0     |        mov %edi, %ebx
10         or     r0,r2,r2     |        mov %esi, %edi
11         bdnz   label2       |        loop label2
12 label3: or     r0,r2,r2     |label3: mov %esi, %edi
13 label1: or     r3,r0,r0     |label1: mov %edi, %esi
14         blr                 |        ret
```

Both programs have the same number of instructions, every PowerPC instruction corresponds to one i386 instruction. One thing is special about this example though: The bdnz

instruction (11) on the left is actually a composed instruction which decrements the count register and does a conditional jump afterwards. The i386 has a very similar instruction, ”loop”. An instruction based recompiler would have to translate ”bdnz” into a decrement and a conditional jump instruction, because ”loop” can only decrement the ECX register, and the decision what register (or memory location) ctr gets mapped to has already been made. This is the single line in this example that might have benefitted from intermediate code, although it would have taken a lot of additional work to be able to map ctr to ECX.
There is one typical kind of code though that could be optimized using intermediate code: The PowerPC cannot load a 32 bit constant into a register, because the whole instruction is only 32 bits long, so it has to be loaded in two steps, like this:

```
lis r27,0x98ba
ori r27,r27,0xdcfe
```

or like this, using another register:

```
lis r30,0x98ba
ori r27,r30,0xdcfe
```

Given, in the second example, r30 is dead after the second instruction, these instructions can be combined and translated into this:

```
mov $0x98badcfe, %esi
```

Unfortunately, a lot more logic is necessary to do this in practice, as the two instructions that belong together might not be subsequent:

```
lis r30,0x98ba
lis r10,0x1032
lis r2,0x6745
lis r4,0xefcd
ori r27,r30,0xdcfe
ori r23,r10,0x5476
ori r29,r2,0x2301
ori r28,r4,0xab89
```

A full data flow analysis would be necessary to optimize this code. All in all, code like this is probably not frequent enough so that an optimization would have a noticeable impact on the overall performance.
The second general advantage of intermediate code is that it makes it possible to work on code, and optimize it in several passes. But this is mostly a point when translating high level languages, because a lot can typically be simplified and optimized in this case. Compiled code already has all redundant code eliminated.
Although it might be an advantage for many designs to be able to easily replace the front ends and back ends with other CPU architectures, this is not necessary for this project, as the whole idea is to design an interface optimized for RISC to CISC translation.

The fourth advantage mentioned above stated that intermediate code can be enriched with information that is not available in the source code but has been reconstructed by an analyzing pass. But this is just one more thing that can be done if you already have intermediate code; it is no argument why intermediate code should be used. This information can just as well be stored in additional data structures.
Intermediate code between the PowerPC and the i386 is not completely useless, but it only makes sense if maximum performance has to be achieved, no matter how much more complex the recompiler gets. The central idea of this project however is a fast recompiler combined with an optimizing recompiler, so intermediate code is not suited for this application.

3.3.8.5 Pass 2 Design Overview

Consequently, pass 2 is supposed to improve the quality of code using the following methods:

- liveness analysis and dynamic register allocation, including condition code optimization
- function call translation and link register elimination

Unfortunately, the detection of atomic functions might not be possible if parts of the program are written in assembly instead of being compiled code, because external code might jump into functions, or functions are not contiguous. Therefore pass 2 is only suited for compiled code.

3.3.9 Dynamic Register Allocation

In pass 1, because of the timing requirements, only static register allocation was possible. As translation speed is not an issue in pass 2, methods can be considered that lead to better code, even if they are more expensive.

3.3.9.1 Design Overview

The LRU method for register allocation was unsuitable for pass 1, as translation would have required significantly more time than other methods if the produced code was supposed to be really better. In pass 2, this method could be reconsidered. The bigger the blocks, the better the produced code, so the LRU method could be used with very big blocks in pass 2. But there is another method for register allocation: It is more expensive than LRU, but typically produces results that are very close to the optimum. This method is used by modern compilers to map local variables to registers, so it can just as well be used to map source registers to target registers. It is done by analyzing the liveness of each source register, in order to find out when it is used and when it is not used, and using an interference graph that represents the information what source registers exist at the same time, a dynamic register allocation is found.

As it is true for all dynamic register allocation strategies, the size of blocks is also important for this method. If basic blocks are analyzed, this method makes no sense, because, as with the LRU method, all registers used have to be read from memory at the beginning of the block, and all registers written have to be written back to memory at the end. Compilers for high level languages use functions as blocks. It makes sense to apply the register allocation algorithm on functions in this project as well, because function calls are supposed to be converted as well, so pass 2 has to do some analysis of the structure of the program in concerns of functions.
So every function has its own mapping of source registers to destination registers. These different mapping have to be synchronized between functions, which means that all important source registers have to be passed in the register array in memory than in target registers.

3.3.9.2 Finding Functions

In order to be able to optimize code in pass 2, all code that belongs to the block that is supposed to be translated has to be found, that is, all code belonging to a function. Searching for the beginning of a function is not necessary if pass 1 only passes basic blocks that are executed by a ”bl” instruction, so only a beginning of a function can be passed to pass 2.
The following algorithm will be used to find all instructions that belong to a function: The code flow inside the function will be completely tracked, that is, all control flow instructions other than ”bl” will be followed. The instruction with the highest address is the last instruction of the function, and since functions are assumed to be contiguous, all data between the first and the last instruction must be code belonging to this function.

3.3.9.3 Gathering use/def/pred/succ Information

The register allocation algorithm requires one pass over the complete code of the function to gather metadata. For every instruction, it has to be known

- the set of registers that are read (”use”)
- the set of registers that are written (”def”)
- the set of instructions that precede this instruction (”pred”)
- the set of instructions that succeed this instruction (”succ”)

The registers in question are r0, r2 to r31, ctr and cr0 to cr7. Compares and instructions with the ”.” suffix write into condition registers, and conditional branch instructions read them.
As the i386 ESP register cannot be practically used for anything else than pointing to a stack, the PowerPC register r1 will be mapped to ESP even when doing dynamic register allocation, so no information will be gathered on r1. The eight (four bit) condition registers cr0 to cr7 are treated like all other 32 bit registers.

This analysis can be done by the run over the code that finds all instructions that belong to the function, so no separate run is necessary. In order to gather this information, all instructions have to be dispatched and decoded.

3.3.9.4 Register Allocation

As soon as every instruction is associated with the use, def, pred and succ, the common register allocation algorithms can do the rest. At first, liveness analysis has to be done. For every instruction, a new set is created (”in”) that contains all source registers that are alive just before this instruction, using the use/def/succ/pred sets. These ”in” sets are then used to create the interference graph, which has one node for every source register that is used, and an edge between two nodes if the two source registers are alive at the same time, i.e. they are in the same ”in” set for any instruction.
The actual register allocation algorithm uses the interference graph to create the final mapping of source registers to target registers or memory, so that source registers that are not alive at the same time can be mapped to a single target register. A parameter of the register allocation algorithm is the number of target registers there are. In this case, this number is 5: EAX and EDX are still scratch registers, and ESP represents the PowerPC stack pointer r1. The mapping targets as returned by the register allocation algorithm, which are therefore between 0 and 4, must of course be converted into the index numbers of the i386 registers that are used for the dynamic mapping: 1, 3, 5, 6 and 7, for ECX, EBX, EBP, ESI and EDI. Source registers that could not be mapped to target registers will be mapped to the corresponding entry in the register array in memory.

3.3.9.5 Signature Reconstruction

Before the function can be retranslated using the optimized register mapping, it has to be found out what source registers are parameters to the function and what source registers are return values. This information, the signature of the function, has been available in the source code of the application, but it has been lost after compilation into binary code. It is required to find out what registers have to be read from the array in memory at the beginning of the functions, and what registers have to be written back at the end. For example, a register that gets written to first does not have to be read from memory before and a register that only gets read does not have to be written back at the end.
Finding out the input registers is simple: All registers that are alive before the first instruction of the function are inputs. These registers get read before they get written to, so they must have been assigned by the caller. But there is no simple way to find out the real return registers: All registers that have been modified, or even registers that are inputs, but not modified, can be return values. The following example demonstrates this:

```
1     cmpwi r3,0x1
2     ble label1
3     ori r4, r3, 1
4     add r3, r3, r4
5 label1:
6     blr
```

The only parameter of this function is r3. If r3 is 0 or 1, the same value will be returned, else r3 + (r3 — 1) will be returned. In this example, there is no way to find out the original meaning of the return value. r3 or r4 or both could contain return values. If instruction 4 is omitted, r3 does not get written any more - but r3 can still be a return value, as the following example shows:

```
 1     blr
```

This function does nothing - but it does preserve all registers, all registers could be parameters and all registers could be used for return values. Or the function has no parameters or return values at all. The following C function will be translated into ”blr” for example:

```
int test(int a) {
    return a;
}
```

There are two ideas about what to write back to the register array in memory: Either write back all registers that are parameters or modified within the function, or filter this set using the PowerPC calling conventions: So only r3 and r4 can contain return values, but this might not be compatible with hand optimized assembly code that uses more registers than r3 and r4 to return data to the caller. Whether all or only some registers are written back is no basic design decision, but an implementation detail, which can be turned on and off at compile or run time.
So all register mapped source registers that are parameters have to be read into target registers, and all register mapped source registers that are used for return values have to be written back into the register array in memory.

3.3.9.6 Retranslation

With the dynamic register allocation and the signature of the function known, the function can be translated. At the very beginning, code has to be emitted that reads all register mapped inputs into i386 registers. If r3 is mapped to ECX, for example, this simply looks like this:

```
    mov register+3*4, %ecx
```

The pass 1 instruction encoder can be used for this. For the translation of the function body, the complete pass 1 recompiler can be used, as most of the work that has to be done is identical. There are just two exceptions: The i386 converter has to use the dynamic register mapping that has been gathered for this function, instead of the static pass 1 mapping. And there are some differences for certain instructions. Every time a ”blr” instruction is encountered, the i386 ”call” instruction can now be used, but all register mapped registers have to be written back to memory before, and read back from memory afterwards. Actually, not all registers have to be saved, but only registers that are not used by the function. As most functions will use all of the few i386 registers and a recursive algorithm has to be used to find out what registers are used in a function and its subfunctions, it is probably not worth bothering.

3.3.9.7 Address Backpatching

Branch instructions have to be translated differently in pass 2 as well, because the situation is completely different. In pass 1, branches were only possible at the end of a block. The target addresses always were the beginnings of basic blocks. If the addresses were not known yet, the last instruction of this block could be changed later.
Now branches can occur in the middle of a translation unit (a function). The target address in case the branch is taken is an address within this function, and as the translation of the next source instruction (not-taken case) always follows the branch, there is no need to handle the not-taken case. The branch instruction therefore has to be filled with the i386 address of the target code, so the recompiler has to create a mapping table from PowerPC to i386 code addresses.
The target address can either be a higher or a lower address than the current address in the source code. In case it is a lower address, the i386 equivalent is already known and can be used to encode the branch instruction. If it is a higher address, the i386 address is not known yet, as the function would have to be translated completely for the code address mapping table to be complete.
This is also a common problem in compiler construction, and there are two solutions:

- 2 pass translation: The code gets translated twice. In the first pass, no code gets emitted, but the mapping table is created. In the second pass, the mapping is completely known and all code can be emitted.
- backpatching: The code gets translated once. All jump target addresses that are not yet known will be left blank and for all of these, the address of the instruction as well as the source address that could not be mapped will be noted. When translation is complete, with the help of the notes, the blank addresses will be filled.

The backpatching method is considerably faster, and its implementation is only marginally more complex, therefore it makes sense to favor this method.

3.3.9.8 Condition Codes

Saving condition registers in i386 registers is simple. The register allocation algorithm will also consider the eight condition registers and might map them to i386 registers. When actually translating an instruction that accesses a condition register, different code can be emitted in pass 2, if the condition register is register mapped. Given cr0 is mapped to ECX, a compare/branch combination could look like this (signed).

```
    cmp $5, %esi
    lahf
    mov %ah, %ecx
[...]
    test $0xc000, %ecx
    jnz label1
    nop
```

In the case of unsigned values, the register move can be integrated into the ”lea” instruction:

```
    cmp $5, %esi
    lahf
    lea (%eax,%eax), %ecx
    rcr $1, %cx
[...]
    test $0xc000, %ecx
    jnz label1
    nop
```

With the condition codes mapped this way, branches should not be significantly slower on the i386 than in native code.

3.3.10 Function Call Convertion

The first analysis of the code in pass 2 already had to deal with whole functions, so it is relatively easy to add more functionality that works on functions. The ”bl”/”blr” style function calls are supposed to be translated into ”call”/”ret” style, since this is faster on an i386.

3.3.10.1 ”call” and ”ret”

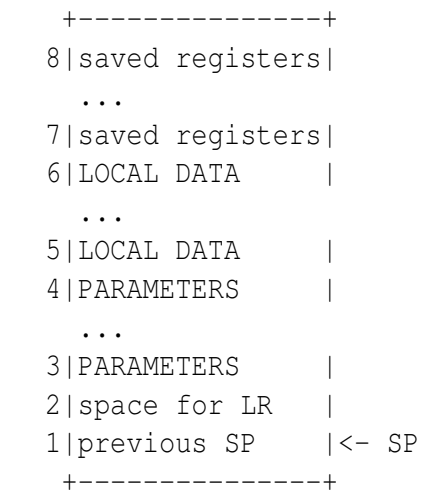
”bl” might just be naively translated into ”call” and ”blr” into ”ret” - but this breaks most programs. The ”call” instruction writes a 32 bit value on the stack and thus modified the stack pointer. If there are more than eight parameters, these are passed on the stack, in the caller’s stack frame. The called function will read them from the caller’s stack frame, but if the return address had been additionally written onto the stack, between the current the the old stack frame, all these accesses are off by 4 bytes. Two solutions come to mind:

- patch the displacements that access the caller’s stack frame
- store the return address in the ”old stack pointer” field on stack

The first solution would detect all instructions in the function that access values in the caller’s stack frame and add 4 to the displacement. It is not trivial to find out what stack accesses target the caller’s stack frame: These are accesses that have a displacement larger than the size of the current stack frame, so the size of the current stack frame has to be found out.
The second solution does not need to adjust stack accesses. Instead, the return address will not be placed in addition into the stack, but it overwrites the uppermost value, by adding 4 to the stack pointer before the call, and subtracting 4 afterwards. The value that gets overwritten is the saved old stack pointer, as shown in figure 3.9.
The previous stack pointer is rarely used: Most code does not destroy the stack pointer by reading the old value from the stack, as shown here:

```
    lwz r1,0(r1)
```

Figure 3.9: PowerPC Stack Frame



but by adding the size of the stack frame to the stack pointer, which saves a memory access:

```
addi r1, n
```

Some applications still use the first method, in which case this instruction has to be replaced with an ”add” during translation. For this, the size of the stack frame has to known, just like above.
All in all, the complexity of both methods is about the same. The first method looks cleaner and is faster, the second one looks more compatible. For the sake of code speed, the first method will be implemented.

3.3.10.2 Stack Frame Size

For adjusting accesses to the caller’s stack frame, the stack frame size of the current function has to be known. This is pretty easy to find out: In the code analysis in pass 2, which looks for the extent of the function and gathers use/def/pred/succ data, can also find out the size of the stack frame. There should be only one instruction like this in the code:

```
stwu r1, -n(r1)
```

The constant of n is the size of the stack frame.

3.3.10.3 ”lr” Sequence Elimination

All PowerPC functions that call one or more functions themselves have to save the link register, because the ”bl” instruction would overwrite it otherwise, and the function would have no way to return to the caller. According to the PowerPC EABI, the link register is stored on the stack, in a field in the stack frame of the caller. There is no way to write the link register into memory directly, it has to be copied into a GPR first. The prolog and the epilog of a function roughly look like this:

```
1    mfspr r0,lr          ; get link register
2    stmw r30,0xfff8(r1) ; save r30 and r31 on stack
3    stw r0,0x8(r1)       ; save link register into caller's stack frame
4    stwu r1,0xffb0(r1)  ; decrease stack pointer by 0x30 and save it
[...actual code in the function...]
5    lwz r1,0x0(r1)       ; restore stack pointer
6    lwz r0,0x8(r1)       ; get link register from caller's stack frame
7    mtspr lr,r0          ; restore link register
8    lmw r30,0xfff8(r1)  ; restore r30 and r31
9    blr                  ; return
```

When translating this code into i386 code, the instructions 1, 3, 6 and 7 need not be translated, as there is no link register, so nothing has to be saved or restored. It is easy to omit the ”mfspr” and ”mtspr” instructions as they are easy to detect. It is not as easy to detect the instructions 3 and 6 in this example. The link register will be placed at the position ”stack pointer + 8” - but only if the stack frame has not been created yet. Otherwise, the address will be ”stack pointer + stack frame size + 8”, which is often the case. The same problem exists when restoring the link register: The stack frame could either still exists or already be destroyed. If the stack frame still exists, for example, the address ”stack pointer + 8” does not point to the the saved link register.
So when translating a function, it must be known at every time whether the stack frame exists or not. In this example, instruction 3 will be detected as the lr save instruction, because there has been no ”stwu r1, -n(r1)” before. And because the stack frame has already been destroyed in instruction 5 (”lwz r1,0(r1)” - could also have been ”addi r1, n”), instruction 6 will be detected as the instruction that loads the link register. The tracking of the existence of the stack must be done during the actual translation, i.e. the instruction recompiler must provide this information.
There is one problem to this: A function may have multiple exits and therefore multiple sequences to destroy the stack frame. After the first ”blr”, the stack frame is considered nonexistent, but it probably exists. In this case, later restore sequences will not be properly recognized, but this is no major problem, as it just leads to unnecessary code.
Because the instructions 1, 3, 6 and 7 will never be translated, no use/def information should be gathered from these instructions. Otherwise some registers would be unnecessarily marked alive and the dynamic register allocation would be less optimal.

Chapter 4

Some Implementation Details

> ”This is no computer, this is my arch enemy!” - *Chief O’Brien, ”The Forsaken”, Star Trek Deep Space 9*

In the context of this project, a recompiler, called ”ppc_to_i386”, has been implemented that is based on the design described in the previous chapter. The implementation has been targeted for inclusion as the recompiler engine in the SoftPear [28] project. It runs on GNU/Linux and FreeBSD. The current status is that not all instructions have been implemented, so only specific test programs will run instead of real world applications. But the infrastructure is basically complete and well-tested, so completing the recompiler should only be a matter of implementing the remaining opcodes.

4.1 Objectives and Concepts of the Implementation

The main objective of this project is to develop ideas and concepts for RISC to CISC recompilation. Since a recompiler is an extensive project, especially if it contains as many ideas as described in chapter 3, it is unrealistic to implement a fully working and tested system in the scope of a diploma thesis. Therefore the main principle of the implementation is to keep complexity low where possible, and implement the important ideas and algorithms instead of all instructions. Some ideas, like branch optimization during register mapping as described earlier, are only possible in an implementation in assembly language. However C has been chosen as the language to write the system in, in order to keep the complexity of the implementation low and to keep the system open for adaptations to other RISC/CISC CPUs. Furthermore, data structures are chosen by their simplicity, not their speed. Also, the design has been changed a little: Pass 2 analysis (”use/def/succ/pred”) will be done by pass 1 translation already, otherwise a lot of code would have had to be duplicated, making development more complex and debugging harder. All simplified parts of the implementation can be changed into more optimized variants without changing the whole structure of the implementation, though. The combined pass 1 translation and pass 2 analysis for example could be split to match the specification better by duplicating the code and deleting all irrelevant code in each copy, or by changing all C code ”if” statements to compile time ”#ifdef” preprocessor statements and compile and link the code twice with different parameters. As a consequence of these simplifications, the speed of the translator (not the

speed of the produced code) is certainly lower than it could theoretically be and as it has been described earlier. As recompilation is a very low-level topic, the design must take into account and be very close to implementation. Many concepts and implementation details have therefore already been described along with the design. Other details are self-evident and need no detailed description - for example, nothing can be said about the pass 1 recompiler loop other than what has already been said. This chapter will therefore only describe the interesting aspects of implementation as well as those parts that slightly differ from the design.

4.2 Loader

Because of the SoftPear project [28], which is supposed to do user level emulation of PowerPC Darwin/Mac OS X on i386 Darwin/Linux/BSD, it was chosen to implement a loader for Darwin Mach-O executables. This loader has been completely implemented by Axel Auweter and Tobias Bratfisch, who also work on the SoftPear project. While the implementation of a loader is not trivial, as it has to handle the quite complex Mach-O executable file format, its purpose and its interface to the recompiler is simple: All sections get loaded into memory, the entry point address gets read, and all this data is passed to the main function. The exact implementation of the loader is irrelevant in the scope of this project—and not very exciting either.

4.3 Disassembler

The first thing that has been done as soon as the loader was complete, was to write a disassembler for PowerPC binary code. Its purpose is to be used by the translation function to print the disassembly of the current instruction on the screen. Before emulator code existed, the code base could be used as a disassembler; later the interface of the disassembler was changed to always decode and print one instruction at a time. Although the PowerPC instruction encoding is very orthogonal, the disassembler does not use tables of strings, but uses actual code to do the work. It consists of a dispatcher for the opcode and a lot of small functions that decode and print one specific instruction. This is necessary because it is the easiest and most flexible way to handle simplified instructions: In case of certain operand combinations, an instruction can have a special meaning, and a different mnemonic should be printed. So the handler for each instruction decodes the parameters and prints the corresponding disassembly. The dispatching and decoding system is identical with that of the instruction recompiler, so it will be described there.

4.4 Interpreter

The specified system has no interpretive pass, instead it always recompiles all code. Yet there exists an interpreter in the implementation, for several purposes:

- Before the implementation of the actual recompiler was started, an interpreter for the most important instructions has been developed. It was used to test the environment (loader, memory etc.) and to study the behavior of PowerPC code in practice, and was invaluable for understanding certain PowerPC characteristics, such as stack handling.
- The interpreter can be easily used for speed comparisons. Although it is implemented in C, it is quite efficient and should not be significantly slower than an optimized implementation in assembly language. Nevertheless, its use is restricted to very simple programs.
- The system still needs an interpreter for all control flow instructions other than ”blr”. These are initially not translated and must therefore be interpreted. Calculated jumps (”bctr”) will currently never be recompiled, so all programs that include jump tables depend on the interpreter.

The implementation is simple. Like the disassembler, it shares the dispatcher and the decoder with the recompiler. The specific function for one instruction implements the behavior of the instruction without simplification, working on the array of registers in memory. Even though the interpreter is mostly only used for a few instructions, the complete interpreter exists in the recompiler code, as interpretation, that is, dispatching, will not be slowed down by the existence of additional instruction implementations.

4.5 Basic Block Cache

The translated i386 code is written into a memory region reserved on the heap. It is currently assumed that it is large enough to hold all translated code. All i386 instructions will be written into this memory region, one basic block or function after another. The layout of the translated code region is managed by two additional data structure, one for basic blocks and one for functions. Both these data structures are linked lists, which have been chosen because of their simplicity. One node of the basic block linked list represents one basic block and contains the start and end addresses of both the PowerPC and the i386 code, the number of executions, the PowerPC addresses of all successor blocks and some additional flags. A node of the linked list for functions represents one function and contains its addresses, pointers to all subfunctions, its signature and its stack frame size.

4.6 Basic Block Linking

Most of the data stored in a node of the linked list for basic blocks is needed for the linker to be able to quickly retrieve data about successor blocks, about where to find the last instruction of the original block and about where to put the link in the translated block. Every time a block is about to be executed or just has been executed, the basic block linker checks whether it can be linked with its successor block or blocks. All of the following conditions are checked in this order:

- The block is not yet linked (flag of the node).

- The block has at least one successor blocks (field of the node).
- The first target has been recompiled (the node has to be found in the linked list).
- If there is a second target: The second target has been recompiled (as above).

If all these conditions are met, a small linker function gets called, in which a simple C switch construct does the dispatching, decoding and encoding. The i386 instructions will be written at the address of the end of the basic block as found in the basic block node, overwriting the old jump back to the recompiler loop.

4.7 Instruction Recompiler

The instruction recompiler in "ppc_to_i386" is both a very efficient and a very structured and readable implementation. All functions are separated into dispatcher (recompile()), decoder (rec_*()), register mapping (VREG()), i386 converter (v_put_*()), instruction encoder (put_*()) and utility functions.

4.7.1 Dispatcher

The dispatcher (recompile()) extracts the upper 6 bits of the instruction code and jumps to the specific handlers using a jump table. Some of these handlers just print error messages for undefined opcodes, some are functions of the decoder, and the handlers for the opcodes 19 and 31 are again functions that jump to more handlers using a jump table, as the primary opcodes 19 and 31 have an extended opcode field. So there are three jump tables, one for the primary opcode, one for the extended opcode if the primary opcode was 19, and one for the extended opcode if the primary opcode was 31. These tables occupy roughly 2 KB, and should not be a problem for the CPU cache. For optimization purposes, no explicit parameters are passed to the decoder functions, as all data that is relevant to the decoder is in global variables.

4.7.2 Decoder

The decoder consists of one function for every PowerPC instruction, whose name is with rec_, followed by the name of the PowerPC instruction. Decoding has been implemented in a readable and efficient way by using preprocessor macros. The PowerPC architecture defines names for certain bit ranges in the instruction code, such as rA for the first register operand and UIMM for an unsigned immediate value. The macro to extract a bit range from a 32 bit value looks like this:

```
#define bits(data, first, last) ((data >> 31-last) &\
    ((1 << (last-first+1))-1))
```

There is one macro for every named bit range in the instruction code. These look like this:

```
#define rA bits(c,11,15)
#define rB bits(c,16,20)
#define UIMM bits(c,16,31)
#define SIMM ((signed short)bits(c,16,31))
```

They always pass ”c” to the bits() macro, which is a global variable that contains the instruction code. It is now easy to access any field in the instruction code just by using its name, like this (taken from the implementation of ”or”):

```
if (rA == rS && rA == rB) { // "mr."
    if (!Rc) return; // nop
    v_put_test_register(VREG(rA));
}
```

The compiler will fill rA, rS, rB and Rc with the specific macro bodies and make sure that only fields are extracted that are actually used. Apart from extracting, the decoder has to detect special operand combinations and, depending on the result, to call functions of the i386 converter. Detecting special combinations of operands is easy, as can be seen in the example above. This sequence checks whether all three operands of the or operation are the same. If they are, the Rc bit is tested. If it is cleared, the instruction is effectively a no-operation, and no code will be emitted. Otherwise, the i386 converter will be called to produce code for a register test of the given register operand.

4.7.3 Register Mapping

Register mapping is supposed to be done in the i386 converter, because the decoder should be as independent of the target architecture as possible. Nevertheless, in the implementation, it is technically already done in the decoder, although in a way independent of the target architecture. There are three data types that represent registers: sreg, vreg and treg.

- an sreg (source register) is an integer that has a value from 0 to 41, representing a PowerPC register. 0 to 31 are the GPRs, 32 is lr, 33 is ctr, and 34 to 41 are cr0 to cr7.
- a vreg (virtual register) is an integer pointer (int*). It can either have a value from 0 to 7, representing an i386 register, or be a pointer into memory, representing a memory location that contains the value of a memory mapped register.
- a treg (target register) is an integer that has a value from 0 to 7, representing an i386 register. The numbering is the native numbering inside the CPU.

All register numbers extracted from the instruction code are sregs. They are converted into vregs when the decoder calls the i386 converter. vregs in turn are either converted into integer pointers or tregs between the i386 converter and the instruction encoder. So the register conversion actually takes place between the layers, so that the decoder can always work with sregs, the i386 converter with vregs, and the instruction encoder with tregs. This makes code easier to read, as the conversion is done quite transparently by making macros convert the registers on the function call:

```
        v_put_test_register(VREG(rA));
```

This code taken from a decoder calls a function of the i386 converter. The register rA is converted from an sreg into a vreg by the macro VREG(), which effectively converts the PowerPC register number into an i386 register number or a memory location, according to the static or dynamic mapping table, and returns it as a vreg. The function v_put_test_register() takes a vreg as an argument, as the signature shows:

```
void v_put_test_register(vreg r);
```

Conversion from vregs to tregs or memory locations is easier. The vreg already contains all information. If a vreg contains an i386 register number and is therefore in the range of 0 to 7, it can just be casted into a treg. If it is not, it can be casted into an integer pointer (int*). The test, whether a register is memory mapped, that is, a vreg is a treg, is done by the macro VREG_IS_TREG(), which checks whether the value is below 8 and returns either true or false.

4.7.4 i386 Converter

All functions of the i386 converter start with v_put_ and implement one abstract operation. A function must test all register operands (which are in vreg format) whether they are register or memory mapped and, according to the result, call functions of the instruction encoder to produce the optimal sequence of code, or different functions of the i386 converter for more complex operations. The following code is the implementation of a register move:

```
inline void v_put_move_register(vreg r1, vreg r2, treg temp, int flags) {
    if (VREG_IS_TREG(r1) && VREG_IS_TREG(r2)) {
        put_move_register((treg)r1, (treg)r2);
        if (flags) put_do_flags((treg)r1);
    } else if (VREG_IS_TREG(r1) && !VREG_IS_TREG(r2)) {
        put_store((treg)r1, r2);
        if (flags) put_do_flags((treg)r1);
    } else if (!VREG_IS_TREG(r1) && VREG_IS_TREG(r2)) {
        put_load(r1, (treg)r2);
        if (flags) put_do_flags((treg)r2);
    } else {
        put_load(r1, temp);
        put_store(temp, r2);
        if (flags) put_do_flags(temp);
    }
}
```

This example has two register parameters and must therefore decide between four cases. In case both registers are register mapped, they are casted into tregs and the function put_move_register() of the instruction encoder is called with will emit a single i386 register

move instruction. In case the first register, which is the source register of the move, is register mapped, but the second one is not, this corresponds to a memory store, so put_store() will be called. If the first operand is memory mapped and the second one is register mapped, it is a memory load. In case both registers are register mapped, the first one will be read into a temporary register and the temporary register will then be stored into the second register in memory. This temporary register is a parameter passed by the decoder. If the decoder translates a complex sequence, which uses both temporary registers, it can make sure that the parts of the i386 converter it calls do not overwrite each other's temporary registers. The boolean parameter "flags" has also been passed by the decoder. If it is true, the i386 converter has to make sure the emulated condition codes are updated, and therefore calls the instruction encoder function put_do_flags(). This whole function is very similar to many other functions in the i386 converter, but unfortunately, it cannot be generalized, because the i386 is very complex and there are many exceptions. This could be ignored in many cases, but the code will be a lot better if it is not, and translation will not be slower.
Nevertheless some functions help recode duplicate code: It is often necessary to have the value of a PowerPC register in an i386 register, no matter if it is memory or register mapped, for example, in order to use it as an index in an indexed addressing mode. The function v_get_register() takes a vreg and an i386 scratch register as an argument, and always returns a treg: If the PowerPC register is register mapped, it will return the vreg from the parameter, casted to a treg, and if it is not, the value will be read into the scratch register and the scratch register will be returned. Functions that write into a register can use the function v_find_register() which again returns either the given vreg or the scratch register. Afterwards, the function v_do_writeback() writes the value into memory, if the PowerPC register is memory mapped, or just does nothing, as the code already has written the value into the correct i386 register.
The following code, taken from the implementation of the 3 register addition, illustrates the usage of v_find_register() and v_do_writeback():

```
vreg r3_temp = v_find_register(r3, TREG_TEMP1);
v_put_move_register(r1, r3_temp, TREG_NONE, NO_FLAGS);
v_put_add_register(r2, r3_temp, TREG_NONE);
v_do_writeback(r3, r3_temp);
```

r1 and r2 are supposed to be added, and the result is supposed to be written into r3. After the first instruction, r3_temp will either be assigned to the index of the i386 register that r3 is mapped to, or to the index of the the first scratch register (EAX), if r3 is memory mapped. Then r1 gets copied into r3_temp and r2 gets added to r3_temp. Both calls start with v_put_, so they produce valid code no matter whether r1 and r2 are register or memory mapped. The fourth instruction will then write EAX back into memory, if r3 is memory mapped, or just do nothing if r3 is register mapped and the last two operations already wrote into the correct i386 register.

4.7.5 Instruction Encoder

Every function of the instruction encoder starts with put_ and takes sregs and register parameters. There is one function for every instruction/addressing mode combination. As

only a small subset of the i386 addressing mode capabilities are used when translating from PowerPC code, the instruction encoder has a manageable number of functions. The instruction encoder detects special cases of an i386 instruction that can be encoded more efficiently: If an addition of a register with an immediate value of 1 is supposed to be encoded, the instruction encoder's function to increment a register is called instead. If the encoding is not to be routed to another function, the optimal encoding of the concrete i386 instruction will be emitted. For example, there are some instructions that have shorter (redundant) encodings if EAX/AX/AL is used as an operand.

4.7.6 Speed of the Translation in Four Steps

All functions of the instruction recompiler are marked as "inline", to make sure that no function will ever be called, instead, every function of the decoder will contain all code necessary to completely translate a PowerPC instruction. The C compiler can do many optimizations: As there are no function calls, there will be no parameter passing; and some calculations like treg tests can be combined into one step. Although every step of the recompiler can have many "if" instructions, some of these can be omitted by the C compiler. The following (although theoretical) example illustrates this: If the decoder calls the i386 converter, which is supposed to translate an addition of the constant register r3 and the constant value of 1, there will be no check, whether the PowerPC register is register or memory mapped: The C compiler has already omitted this check, as the register r3 is constant. The instruction encoder would then test whether the immediate is 1, but this is known at compile time as well. So the C compiler would effectively only produce code to emit "inc %eax".

4.7.7 Execution

The function that is supposed to pass control to recompiled code and that provides a reentry point for the recompiled code into the recompiler loop is a bit tricky. If it is not completely correct, it will just not work at all. This function has to save all i386 registers and load the mapped ones. On reentry, it has to do the reverse. The i386 registers are saved using the pusha instruction[1], then ESP is stored in a variable in memory. After the mapped registers have been loaded from the register array in memory, this function jumps into the recompiled code, at the address that was passed as a parameter. When the recompiled code wants to jump back, it jumps to the instruction in this function that is just after the jump into the recompiled code. Now the EAX register, which holds the address where the recompiled code stopped execution, will be written into a variable, all mapped registers will be stored into the register array, the stack pointer will be restored, and the remaining i386 register will be read from the stack. The function then returns to the caller.

[1] It is an interesting trivia why "pusha" also pushes ESP

4.8 Dynamic Register Allocation

The pass 2 analysis of the code, which should be done in a separate run over the code as soon as a function is to be compiled by pass 2, is already done by the pass 1 recompiler in the implementation, for simplicity purposes. Every time the VREG() macro would be used in the decoder, either VREG_R() or VREG_W() is used instead, which, besides register conversion, also associates the current instruction with the information that the register that is supposed to be converted is either read or written. This creates the use[] and def[] arrays. The pred[] is actually not necessary for the liveness analysis algorithm, so only the succ[] array gets created. One item of the use[] and def[] arrays, which is a set of registers, is implemented using a 64 bit integer variable, each bit representing one source register. As there are only 42 source registers, 64 bits are more than enough. Liveness analysis, interference graph construction and register allocation will then be done using the standard algorithms. A node of the interference graph is also implemented using a 64 bit integer. Every bit represents an edge to one of the other registers.

Chapter 5

Effective Code Quality and Code Speed

As it has been said earlier, the main purpose of the implementation is the verification of the design. The implementation shows how well the design works in practice and how good the produced code is. It is not the purpose of the implementation to provide a platform for measurements of the translation speed, because the two objectives ”verification” and ”speed” are somewhat mutually exclusive, unless there is a lot of time for the implementation.
It might be argued that it makes little sense to write a recompiler whose new ideas target high speeds, but not optimize the implementation for speed. But this point ignores that there are two kinds of speed: the speed of translation, and the speed of the translated code. The maximum speed of the pass 1 translation has been theoretically calculated during the design phase; an optimized implementation in assembly language should basically reach this speed. The speed of the pass 2 recompiler does not really matter: The idea is to run it only on code that is heavily used, so that even complex pass 2 algorithms will amortize quickly.
The performance of the recompiled code is more interesting. This speed can either be measured, or it can be evaluated by inspecting the quality of the code. This chapter will therefore concentrate on the quality of the code produced by the pass 1 and 2 algorithms, as well as on speed measurements of the translated code in pass 1.

5.1 Pass 1

On the PowerPC, all programs in this chapter have been compiled with ”gcc (GCC) 3.3 20030304 (Apple Computer, Inc. build 1495)” on Mac OS X 10.3. On the i386, the test programs have been compiled with ”gcc (GCC) 3.3 20030226 (prerelease) (SuSE Linux)” on SuSE Linux 8.2.
All programs in this chapter have been compiled with -O3 on both platforms. The PowerPC disassemblies in this chapter have been done with Mac OS X ”otool -tv” . The recompiled machine code gets written into a headerless file by the recompiler, which can be disassembled in Linux using ”objdump -m i386 -b binary -D”.
The test program has been written in C and only contains a small, artificial function that takes many minutes to execute. This way, the recompiler does not have to be able to translate many PowerPC instructions, and the measurement will not be influenced by the

recompilation speed too much. The test program is a recursive implementation of the Fibonacci function. The C code looks like this:

```
unsigned int __attribute__((noinline)) fib(unsigned int i) {
        if (i<2) return i;
        else return fib(i-1) + fib(i-2);
}

int main() {
        return fib(50);
}
```

The ”noinline” attribute makes sure that the first recursion of fib() does not get embedded into main(), which leads to more readable code. The complete PowerPC version looks like this:

```
_fib:
00001eb0        cmplwi  r3,0x1
00001eb4        mfspr   r2,lr
00001eb8        stmw    r29,0xfff4(r1)
00001ebc        stw     r2,0x8(r1)
00001ec0        or      r30,r3,r3
00001ec4        stwu    r1,0xffb0(r1)
00001ec8        ble+    0x1ee4
00001ecc        addi    r3,r3,0xffff
00001ed0        bl      0x1eb0
00001ed4        or      r29,r3,r3
00001ed8        addi    r3,r30,0xfffe
00001edc        bl      0x1eb0
00001ee0        add     r3,r29,r3
00001ee4        lwz     r4,0x58(r1)
00001ee8        addi    r1,r1,0x50
00001eec        lmw     r29,0xfff4(r1)
00001ef0        mtspr   lr,r4
00001ef4        blr
_main:
00001ef8        li      r3,0x32
00001efc        b       0x1eb0
```

5.1.1 Code Quality

The instructions are very much interleaved and the code is not easy to read. The i386 code is naturally even worse, therefore it is discussed block by block. Note that the addresses are not in the order of the original code, but in the order of first execution. Also note that the code actually resides at 0x40160000 in memory in this example; so this base address

is visible in all absolute addresses, in this case all addresses loaded into the emulated link register.

```
00001ef8        li      r3,0x32
  1b:   be 32 00 00 00          mov    $0x32,%esi
00001efc        b       0x1eb0
  20:   b8 fc 1e 00 00          mov    $0x1efc,%eax
  25:   ff 25 ec d5 05 08       jmp    *0x805d5ec
```

This is the main() function. As it only gets executed once, it never gets linked to fib(). Instead, the code loads the address of the instruction that has to be interpreted into EAX and returns to the interpreter.

```
00001eb0        cmplwi  r3,0x1
  3b:   83 fe 01                cmp    $0x1,%esi
  3e:   9f                      lahf
  3f:   8d 04 00                lea    (%eax,%eax,1),%eax
  42:   d0 dc                   rcr    %ah
  44:   88 25 10 d6 05 08       mov    %ah,0x805d610
00001eb4        mfspr   r2,lr
  4a:   89 e9                   mov    %ebp,%ecx
00001eb8        stmw    r29,0xfff4(r1)
  4c:   8b 15 b4 d5 05 08       mov    0x805d5b4,%edx
  52:   89 54 24 f4             mov    %edx,0xfffffff4(%esp,1)
  56:   8b 15 b8 d5 05 08       mov    0x805d5b8,%edx
  5c:   89 54 24 f8             mov    %edx,0xfffffff8(%esp,1)
  60:   8b 15 bc d5 05 08       mov    0x805d5bc,%edx
  66:   89 54 24 fc             mov    %edx,0xfffffffc(%esp,1)
00001ebc        stw     r2,0x8(r1)
  6a:   89 4c 24 08             mov    %ecx,0x8(%esp,1)
00001ec0        or      r30,r3,r3
  6e:   89 35 b8 d5 05 08       mov    %esi,0x805d5b8
00001ec4        stwu    r1,0xffb0(r1)
  74:   89 64 24 b0             mov    %esp,0xffffffb0(%esp,1)
  78:   83 c4 b0                add    $0xffffffb0,%esp
00001ec8        ble+    0x1ee4
  7b:   f6 05 10 d6 05 08 c0    testb  $0xc0,0x805d610
  82:   0f 85 0e 00 00 00       jne    0x96
  88:   e9 42 00 00 00          jmp    0xcf
```

The first five instructions of the i386 version of this basic block implement the comparison. As it is an unsigned comparison, the i386 flags have to be modified so that signed conditional jump instructions will work. The instruction at address 0x44 stores the flags in the memory address that corresponds to cr0. Instruction 0x4a copies the link register (EBP) into r2 (ECX). It will be stored into the stack in instruction 0x6a. The code from 0x4d to

0x66 implements the stmw instruction, which stores r29 to r31 onto the stack and must be unrolled on the i386. But this should only be longer, not slower. Unfortunately, r29 to r31 are memory mapped, so every register has to be read from memory and then stored onto the stack.
Instruction 0x6e corresponds to a register move. As r30 is memory mapped, EAX, which is r3, will be stored in memory. The ”stwu” instruction must also be done in two steps on the i386. The conditional branch is expanded into a ”testb” and two branches. It can be seen that this block has been linked to the two succeeding blocks, which have been translated to 0x96 and 0xcf.

```
00001ecc        addi    r3,r3,0xffff
  cf:   4e                      dec     %esi
00001ed0        bl      0x1eb0
  d0:   bd eb 00 16 40          mov     $0x401600eb,%ebp
  d5:   e9 61 ff ff ff          jmp     0x3b
```

The following PowerPC basic block only consists of two instructions. The first one, the addition of -1, is translated into a very effective ”dec” instruction. The ”bl” (function call) is translated into a sequence that moves the address of the subsequent block into EBP and jumps to the function. The successor block is located at 0xeb.

```
00001ed4        or      r29,r3,r3
  eb:   89 35 b4 d5 05 08       mov     %esi,0x805d5b4
00001ed8        addi    r3,r30,0xfffe
  f1:   8b 35 b8 d5 05 08       mov     0x805d5b8,%esi
  f7:   83 c6 fe                add     $0xfffffffe,%esi
00001edc        bl      0x1eb0
  fa:   bd 15 01 16 40          mov     $0x40160115,%ebp
  ff:   e9 37 ff ff ff          jmp     0x3b
```

The first line copies r3 (ESI) into the memory mapped register r29. 0x1ed8 subtracts 2 from r30 and stores the result in r3. The i386 code loads the memory mapped register into ESI and subtracts 2 from it. Note that an immediate ”add” instruction has been emitted instead of a ”sub $2”, because this would neither be smaller nor faster, and just make the recompiler slower. The value of ”-2” is stored as an 8 bit operand that will be sign extended to 32 bit by the CPU at runtime. Instructions 0xfa and 0xff encode a ”bl” instruction again.

```
00001ee0        add     r3,r29,r3
 115:   03 35 b4 d5 05 08       add     0x805d5b4,%esi
00001ee4        lwz     r4,0x58(r1)
 11b:   8b 7c 24 58             mov     0x58(%esp,1),%edi
00001ee8        addi    r1,r1,0x50
 11f:   83 c4 50                add     $0x50,%esp
00001eec        lmw     r29,0xfff4(r1)
 122:   8b 54 24 f4             mov     0xfffffff4(%esp,1),%edx
 126:   89 15 b4 d5 05 08       mov     %edx,0x805d5b4
```

```
 12c:   8b 54 24 f8             mov    0xfffffff8(%esp,1),%edx
 130:   89 15 b8 d5 05 08       mov    %edx,0x805d5b8
 136:   8b 54 24 fc             mov    0xfffffffc(%esp,1),%edx
 13a:   89 15 bc d5 05 08       mov    %edx,0x805d5bc
00001ef0        mtspr   lr,r4
 140:   89 fd                   mov    %edi,%ebp
00001ef4        blr
 142:   ff e5                   jmp    *%ebp
```

The rest of the function is another basic block. The addition of a memory mapped register to the register mapped r3 can be done by an ”add” instruction that has a memory address as an operand. 0x1ee4 reads from the stack into a register, and is translated into the corresponding instruction at 0x11b, just like the addition of 0x50 to the stack pointer (0x1ee8, 0x11f). The ”lmw” instruction reads three registers from the stack, and must again be done in six instructions on the i386, because all registers are memory mapped. 0x1ef0 copies the link register back, and 0x1ef4 returns to the caller. On the i386, this is a jump to the address stored in the emulated link register, EBP.
Because the branch instruction at 0x1ec8 has 0x1ee4 as a target, which is the same as the last basic block without the first instruction, the basic block has to be translated again, from the second instruction on:

```
00001ee4        lwz     r4,0x58(r1)
  96:   8b 7c 24 58             mov    0x58(%esp,1),%edi
00001ee8        addi    r1,r1,0x50
  9a:   83 c4 50                add    $0x50,%esp
00001eec        lmw     r29,0xfff4(r1)
  9d:   8b 54 24 f4             mov    0xfffffff4(%esp,1),%edx
  a1:   89 15 b4 d5 05 08       mov    %edx,0x805d5b4
  a7:   8b 54 24 f8             mov    0xfffffff8(%esp,1),%edx
  ab:   89 15 b8 d5 05 08       mov    %edx,0x805d5b8
  b1:   8b 54 24 fc             mov    0xfffffffc(%esp,1),%edx
  b5:   89 15 bc d5 05 08       mov    %edx,0x805d5bc
00001ef0        mtspr   lr,r4
  bb:   89 fd                   mov    %edi,%ebp
00001ef4        blr
  bd:   ff e5                   jmp    *%ebp
```

The recompiled code clearly suffers from a number of problems:

- There are too many memory accesses.
- Code for compare and branch is too complex.
- Link register emulation is slow.
- There are unnecessary jumps between basic blocks

5.1.2 Speed of the Recompiled Code

The test program has been run in three different environments. The PowerPC binary has first been run in the recompiler, then the i386 binary has been run natively on the same machine, and finally, the PowerPC program has also been run natively on a PowerPC. The i386 machine is a Microsoft Xbox (Pentium III Celeron, 733 MHz), running GNU/Linux, and the PowerPC machine is an Apple iBook G4/800. The speed of the two CPUs should be comparable.
Measurements have been done using the UNIX ”time” tool, which returns the time a program has spent in user mode. As the test program has been designed to run for many minutes, the accuracy of ”time” is sufficient.
Native execution of the i386 code on the Celeron took 1012 seconds for this example. Running the PowerPC version in the emulator took 2152 seconds, which is roughly twice as long. As a comparison, the G4 completed the algorithm in 1192 seonds, The 733 MHz Celeron would therefore be about as fast as a 440 MHz PowerPC G4, which is already pretty good.
The importance of register mapped registers can be shown by running the program again in a recompiler that has been changed to map r16 to r20 (and lr) to i386 registers — these are not used in the program at all, so all registers have to be emulated in memory. The produced code corresponds to the code generated by the most simple recompiler which maps all PowerPC registers to memory. On the Celeron, execution took 3742 seconds, which is almost twice as long as before, and three and a half times as much compared to native code. The Celeron would effectively be a PowerPC G4 with about 200 MHz.

5.2 Pass 2

Pass 2 dramatically improves the code quality. Unfortunately, the implementation does not produce correct code in all cases yet. Therefore another example has been chosen — the *iterative* implementation of the Fibonacci function:

```
__attribute__((noinline)) int fib(int f) {
        int i, a, b;
        int temp;

        if (f<2) return f;

        a = 0;
        b = 1;
        while (--f) {
                temp = a + b;
                a = b;
                b = temp;
        }
        return temp;
}
```

Compiled into PowerPC code, this function looks like this:

```
00001eb0        cmpwi   r3,0x1
00001eb4        or      r0,r3,r3
00001eb8        ble     0x1ee4
00001ebc        addic.  r3,r3,0xffff
00001ec0        li      r9,0x0
00001ec4        li      r0,0x1
00001ec8        beq     0x1ee0
00001ecc        mtspr   ctr,r3
00001ed0        add     r2,r9,r0
00001ed4        or      r9,r0,r0
00001ed8        or      r0,r2,r2
00001edc        bdnz    0x1ed0
00001ee0        or      r0,r2,r2
00001ee4        or      r3,r0,r0
00001ee8        blr
```

As before, the translated i386 code will be presented inline with the original PowerPC instructions.

```
 21b:   51                      push   %ecx
 21c:   53                      push   %ebx
 21d:   55                      push   %ebp
 21e:   56                      push   %esi
 21f:   57                      push   %edi
```

The register pushes are done to save all registers that are modified by this function. There is no equivalent to this in the PowerPC code. Because of the different stack layout, accesses to the caller's stack frame would have to be adjusted, as described earlier, but this function does not access the caller's stack.

```
 220:   8b 3d ac 61 05 08       mov    0x80561ac,%edi
```

The function accepts one parameter (r3), which will be passed in the register array in memory. This i386 instruction reads the parameter into a register.

```
00001eb0        cmpwi   r3,0x1
 226:   83 ff 01                cmp    $0x1,%edi
 229:   9f                      lahf
 22a:   88 25 70 ea 05 08       mov    %ah,0x805ea70
```

This is a compare sequence again. This time it is a signed compare, so the flags do not have to be converted. As mapping of PowerPC condition registers to i386 registers has not been implemented yet, the pass 2 recompiler produced code to store the flags in memory.

```
00001eb4        or      r0,r3,r3
 230:   89 fe                   mov    %edi,%esi
00001eb8        ble     0x1ee4
 232:   f6 05 70 ea 05 08 c0    testb  $0xc0,0x805ea70
 239:   0f 85 2f 00 00 00       jne    0x26e
```

As can be seen, code for condition jumps still works like in pass 1. But this time, there is no jump to the next block in the case the condition jump has not been taken. Instead, the following instruction (at address 0x23f) will be executed.

```
00001ebc        addic.  r3,r3,0xffff
 23f:   4f                      dec    %edi
 240:   9f                      lahf
 241:   88 25 70 ea 05 08       mov    %ah,0x805ea70
```

The ”addi.” instruction has been translated into a ”dec”, followed by a two instructions sequence to save the flags.

```
00001ec0        li      r9,0x0
 247:   31 db                   xor    %ebx,%ebx
00001ec4        li      r0,0x1
 249:   be 01 00 00 00          mov    $0x1,%esi
00001ec8        beq     0x1ee0
 24e:   f6 05 70 ea 05 08 40    testb  $0x40,0x805ea70
 255:   0f 84 11 00 00 00       je     0x26c
00001ecc        mtspr   ctr,r3
 25b:   89 f9                   mov    %edi,%ecx
```

Loading the value of zero is done using the more efficient ”xor” instruction that clears a register on the i386. The count register has been mapped to an i386 register, as the instruction in address 0x25b shows.

```
00001ed0        add     r2,r9,r0
 25d:   89 dd                   mov    %ebx,%ebp
 25f:   01 f5                   add    %esi,%ebp
00001ed4        or      r9,r0,r0
 261:   89 f3                   mov    %esi,%ebx
00001ed8        or      r0,r2,r2
 263:   89 ee                   mov    %ebp,%esi
00001edc        bdnz    0x1ed0
 265:   49                      dec    %ecx
 266:   0f 85 f1 ff ff ff       jne    0x25d
```

This is the loop inside the algorithm. The addition and the register moves were translated quite efficiently, as all these instructions work on registers. Yet, the addition could also have been done using ”lea”, which would have been faster, but this is not implemented yet. The

”bdnz” instruction, which decrements the count register and branches if it has not reached zero has been translated into two instructions; no condition codes are involved when using ”bdnz”.

```
00001ee0        or      r0,r2,r2
 26c:   89 ee                   mov    %ebp,%esi
00001ee4        or      r3,r0,r0
 26e:   89 f7                   mov    %esi,%edi
00001ee8        blr
 270:   89 3d ac 61 05 08       mov    %edi,0x80561ac
```

The last instruction writes the result back into the register array in memory, so that the caller can read it into the i386 register it has this PowerPC register mapped to.

```
 276:   5f                      pop    %edi
 277:   5e                      pop    %esi
 278:   5d                      pop    %ebp
 279:   5b                      pop    %ebx
 27a:   59                      pop    %ecx
 27b:   c3                      ret
```

Finally, the caller’s registers must be restored, and the function can return. Unfortunately, pass 2 is not yet advanced enough to link a complete program together so that it can be run, so the inspection of the generated code must be enough. All that can be done now is compare the translated code to the code GCC produces when compiling the C program. The important part is the loop, which looks like this:

```
.L9:
        leal    (%edx,%ebx), %ecx
        decl    %eax
        movl    %edx, %ebx
        movl    %ecx, %edx
        jne     .L9
```

There are two differences here. The first difference is that GCC makes use of ”lea”, which could be done by the recompiler as well. What is impossible for the recompiler in its current design is the other optimization GCC makes: The decrement and the the conditional jump instruction are separated, which gives the CPU more time to predict the conditional jump and fill the pipeline. The code generated by the recompiler and the code generated by GCC are very similar though and should have roughly the same performance. This means: The speed of the recompiled code can be very close to the speed of native code, using the relatively simple pass 2 optimizations. But this example might not be typical at all. Real measurements would have to be made, but for that, the pass 2 would have to be fully implemented.

Chapter 6

Future

This thesis has addressed many problems of RISC to CISC recompilation, and has presented many concepts and ideas to solve these. But there are still some aspects of RISC to CISC recompilation that have not been observed at all, and the implementation of the recompiler lead to some new ideas which have not been thoroughly tested or compared to other methods, and for which there was not enough time for a complete implementation.
Therefore this chapter describes missing aspects, further ideas and other applications of the technology that has been developed.

6.1 What is Missing

At the current state, the recompiler that has been developed in the scope of this thesis is of little use in practice, as the implementations of many instructions are missing, so that only test programs run right now. Some side effects of certain instructions, such as the carry and summary overflow flags are not emulated at all. Pass 2 recompilation is mostly implemented, but the interface to the rest of the system is not complete.
Also floating point is missing. Efficient recompilation of RISC floating point instructions to i386 floating point (”i387”) might be a diploma thesis of its own, since the i386 floating point stack makes translation very awkward. Even very optimized recompilers that target i386, such as PearPC, only produce threaded code for floating point instructions in the source language.
Finally, the problem to detect whether code might not be compiler generated and incompatible with the methods described has been ignored. This system always assumes that all code is compiled code. The behavior of every algorithm should be tested in case the code is not what the algorithm expects.

6.2 Further Ideas

Many further ideas have been developed during implementation, but many of them have not been pursued further, either because there was no more time, or because they were by far too complex for the scope of this thesis. A project log [29] is available, in which some additional unimplemented ideas have been described in more detail than in this thesis.

This section summarizes the most important ideas which can be a base for future research on this field.

6.2.1 Translation Concepts

An optimal implementation of a pass 1 recompiler as described in chapter 3 translates one source instruction in about 100 clock cycles. On a 1 GHz CPU, this corresponds to 10 million instructions that are translated per second. So pass 1 recompilation of all code that can be statically detected in the application could be recompiled right after loading it into memory. Newly detected blocks would be handled by the dynamic recompiler, and code could be optimized by pass 2 later. The advantage would be that all code could already be linked, so no jumps back to the recompiler (context switches) would be necessary, unless new code is detected. The speed of the pass 1 recompiler could effectively compile the biggest applications in less than a second, and the additional delay is probably not noticeable.
This initial pass 1 translation could also be saved on disk in order to be available the next time the application is run. Saving metadata and recompiled code between runs could also be done for any pass 1 and pass 2 data in the existing recompiler system.
The idea to statically translate most of the code could also be used with the more optimizing pass 2 compiler. An offline recompiler could translate all available code on a system, so that the runtime environment does not need to translate the bulk of it. Such an offline translator could also apply many more optimizations, as it is not time critical at all. It could make use of intermediate code or even use a conventional C compiler as a back end, by feeding it with generated C code, as done by the staticrecompilers group [26].
Static recompilation suffers from the lack of information that is available when the program is not running. Fortunately, there is some more information available that can be used. Libraries for example, and also some applications include relocation information. This information can be used to find jump tables, for example, which must have corresponding relocation entries if the code is relocatable.

6.2.2 Evaluation of Simplification Ideas

Only since recently, the Open Source PowerPC emulator PearPC has been available, which is compatible enough to run Linux/PPC and Mac OS X. PearPC can be an invaluable tool for gathering statistical data of PowerPC code in practice.

- Do condition registers ever contain illegal values?
- Do certain side effects like the summary overflow flag have to be emulated at all[1]?
- What registers are used most? In this environment, libraries will be included into the statistic, and code that is run more often will be counted more often.
- What is the distribution of the execution frequencies of basic blocks?

[1]Sebastian Biallas, the main author of PearPC, has already found out that Mac OS X will boot and run without the emulation of a summary overflow flag.

It should be possible to only make statistics on code that runs in user mode. The data gathered can be used for the design of simplifications and in order to optimize the recompiler for the most frequent cases.

6.2.3 Dispatcher

A PowerPC instruction dispatcher first has to look at the primary opcode. If it is not 19 or 31, this opcode corresponds to a certain instruction. If it is, the extended 9/10 bit opcode field has to be evaluated. An implementation of a dispatcher will therefore need two non-predictable branches until the handler for a 19/31 instruction is reached—and these instructions form the bulk of the instruction set, which makes PowerPC interpretation and recompilation slower than necessary.
However, a hand-optimized sequence in assembly language can reduce the number of non-predictable branches to one, by using the ”cmov” instruction, which is available on all i686 compliant CPUs (i.e. since Intel Pentium Pro (1995) and AMD K7 (1999)).

```
// fetch
mov (%esi), %eax
add $4, %esi

// extract opcodes
mov %eax, %ebx
mov %eax, %edi
and $0xfc000000, %eax
and $0x000007fe, %ebx

// case: no extended opcode
xor %ecx, %ecx
mov %eax, %edx

// case: extended opcode 19
cmp $0x4c000000, $eax
cmovz $64, %ecx
cmovz %ebx, %edx

// case: extended opcode 31
cmp $0x7c000000, $eax
cmovz $64+2048, %ecx
cmovz %ebx, %edx

// add values
add %ecx, %edx

// effective opcode in %ecx
jmp jumptable(%ecx,4)
```

At first, the primary and the extended opcode get extracted—although only the primary opcode might be necessary. In the default case, if there is no extended opcode, the primary opcode and the value of 0 will be copied into the two destination registers. If the primary opcode is 19, the value of 64 and the extended opcode will be written into the destination registers. In case of opcode 19, the value will be 64+2048. At the end, the two values in the destination registers are added, so the result will be a value from 0 to 64+2*2048-1 (4159). Primary opcodes get mapped to the same values, in case of opcode 19, the extended opcode will be mapped to values from 64 to 2111, and in case of opcode 31, the extended opcode will be mapped to values from 2112 to 4159.
The jump table will be a little over 16 KB in size, which can be halved if all handler functions fit into 64 KB, using this code instead of the jump at the end:

```
mov jumptable(%ecx,2), %eax
jmp code(%eax)
```

The costs of this this dispatcher are an estimated 22 cycles on a modern i386 CPU, which includes the pipeline stall caused by the jump.
As this idea has been pretty much last minute, it has neither been thoroughly tested, nor compared with alternatives. The implementation still does the conventional method with two sequential jumps.

6.2.4 Peephole Optimization

Peephole Optimization is a local optimization technique that only takes into account instructions in the immediate proximity of a certain point. At one point, such an optimization has been implemented: The combination of the PowerPC instructions ”srawi” and ”addze” implements a signed division by a power of two. ”sraw” shifts a register right by 1 to 31 digits, and sets the carry flag if one if the bits shifted out was a 1, otherwise it clears the carry flag. ”addze” then adds the carry flag to the register.
On the i386, this is done differently, but also in two steps. In the first step, the divisor minus one is added to the register, and in the second step, the register is shifted, which leads to the same result.
Translating the two instructions independently would produce very inefficient code. The implementation of ”srawi” would be very complex in i386 assembler, but the combination of ”srawi” and ”addze” is trivial to translate.
The implementation of srawi tests whether the next instruction is addze and has the same register as an operand. If this is the case, it emits the optimized i386 sequence for a signed division by a power of two and skips the next instruction.
There may be many more examples of two or three PowerPC instructions that are often used as a combination, and that should be implemented completely differently on the i386. It should make sense to find more of these combinations in typical code, possibly by making automated statistics.
On the other hand, these optimizations can make the recompiler slower, so they might not be useful in pass 1. More investigation of this topic certainly makes sense.

6.2.5 Instruction Encoder

The Athlon and Pentium 4 optimization manuals [37] [38] contain a lot of very useful information on what should be done and what should be avoided. They also provide a very good insight into the inner workings of modern CPUs. The information in these books can be used to make both the i386 converter and the instruction encoder contribute to optimal code.
For example, all multiplications by a constant up to 32 can be done with small sequences of code that bypass the multiplication unit and are therefore a lot faster. Hints like these can help produce better code without decreasing the translation speed, so as much information as possible from these books should be used for the implementation of additional instruction of the recompiler and for optimizing the translation of existing instructions.

6.2.6 Calculated Jumps and Function Pointers

This design of the recompiler has ignored the problem of calculated jumps and jump tables to some degree. Jump tables are generated by a compiler in case of large ”switch” statements, and calculated jumps are used in all applications that make use of function pointers, such as, ironically, recompilers. If an emulator includes an interpreter for jumps, it can easily handle this problem, but execution might be very slow, as there has to be a context switch for every calculated jump instruction.
As some programs make heavy use of these methods, it makes sense to optimize this case to some degree. A simple method to recompile GCC-generated PowerPC switch-style jump tables into native i386 code has been developed, but not implemented. It detects these constructs by setting one of many different flags every time an instruction is encountered that looks like it could contribute to a switch-style jump table jump. If the ”bctr” instruction is detected, and all flags are set in the current basic block, it is assumed that it is the final jump using a jump table, and the information that has been gathered in this basic block will be used to add some i386 jump table code and to convert the addresses in the jump table to i386 addresses. This method has been described more thoroughly in the project log [29].
Unfortunately however, there can always be calculated jumps in the code that cannot simply be translated. The ARMphetamine project [21] developed a method to embed a sequence of code, which does the source to target address conversion, into the recompiled code. This makes context switches obsolete, and by using efficient hash tables, this method might be only slightly slower than native code.

6.2.7 Pass 1 Stack Frame Conversion

The PowerPC call and return instructions, ”bl” and ”blr” could already be translated in pass 1, using the method to overwrite the field on the stack that contains the old stack pointer. So a ”bl” instruction can be translated into a ”add”/”call”/”sub” sequence as described earlier.
If a function wants to read the old stack pointer in order to destroy the stack frame, it has to be replaced by an instruction that adds the stack frame size to the stack pointer. Finding out the size of a stack frame in pass 1 is a bit trickier in pass 2, but with little effort, pass 1

recompilation can mark basic blocks that belong together and route the information about the stack frame size to the block that includes the destruction code.
This is also an idea that should not slow down recompilation, but should dramatically improve code quality.

6.2.8 Register Allocation

6.2.8.1 Adaptive Static Allocation

The statistics that have been made on register usage have shown quite clearly that some registers are used more frequently than others. Most applications use *these* registers most frequently, but a few applications, such as Apple Keynote, are very different in terms of register usage[2].
So it is obvious that it might be a good idea to gather statistics about register usage just after loading and before running a program. An analysis like this should be very fast and not too complicated to implement. This method could be combined with static pass 1 translation of all known code at loading time.

6.2.8.2 Keeping Lives of a Source Register Apart

There is also room for improvement in the dynamic register allocation strategy. The C compiler, which also applied the liveness analysis and register allocation algorithms, might have assigned more than one variable to one PowerPC register. It is inefficient if the recompiler does not keep these two variables apart: A source register might be used for one variable at the beginning of a function, then be dead for a long time, and be used again for another variable at the end of the function—it has two ”lives”.
The register allocation algorithm of the recompiler can use the emulated version of this register for other registers when the PowerPC register that has been mapped to it is dead, but it can not split the two parts into two target registers. The register allocation that will be found might not be optimal due to the fact that two objects were forced to be mapped together.
It is possible to reconstruct the different independent variables that had been mapped to one PowerPC register by tagging the register in every instruction with the address of the instruction that accesses it first within this ”life”. An iterative algorithm that follows the code flow should be able to do this. A first draft for an algorithm is described in the [29].

6.2.8.3 Better Signature Reconstruction

While it is easy to find out the parameters of a function, it is not easy to find out the registers that are used to pass return values. There is one rather complex method that can find out this information. It is based on the idea that a register can only contain a return value if the caller ever reads it after the function call. As callers are free to ignore return values, it is important that all calls of the function have to be investigated.

[2]The six most frequently used registers in Keynote are r3, r9, r0, r30, r1 and lr.

Standard signature reconstruction of the function has to be done first. The output registers (for now these are all registers that have been written to as well as all parameter registers) and the input registers will be the ”def” respectively the ”use” set of the function call instruction in the caller. After liveness analysis has been done for all callers, the real return registers can be found out by merging the sets of registers that are live just after each function call instruction.
The disadvantage of this optimization is that if new code is detected that calls an existing function, it might be the first caller to evaluate a certain return register, so the function will have to be retranslated to write this return value back into memory. Additionally, this method can only be done with global knowledge about the program. Again, the project log [29] contains more details.

6.2.8.4 Global Register Allocation

The register allocation algorithm in this project has been applied to functions. But it can also be applied to a complete program. Liveness of a register can then range across a function call. This effectively completely changes the interface and even softens the borders between functions. In order for this method to be efficient, it is important to keep the lives of a register apart, as described earlier, because every register certainly has many lives within a program.
This idea is probably not very realistic: Global register allocation can only be applied if all code is known, which cannot be guaranteed. Otherwise, there would be no possibility to merge new code other than doing register allocation and translation again.

6.2.8.5 Condition Codes

When mapping PowerPC condition codes to an i386 register, two condition codes can be mapped to a single register if the i386 register is one of EAX, EBX, ECX or EDX, as these can be accessed as two 8 bit registers each. As EAX and EDX are used as scratch registers, the possible registers can only be EBX and ECX. The register usage statistics showed that 94% of all condition code accesses target cr0 or cr7. So the register allocation algorithm could always regard cr0 and cr7 as an entity and merge them in the interference graph, so that they will be mapped to the same register. It must be made sure though that the condition registers are mapped to EBX or ECX only, but this is easy: After the allocation has been done, the register that was mapped to EBX can be swapped with what cr0 and cr7 were mapped to.
Interestingly, combining cr0 and cr7 in the register usage statistics would lead to position 6 in the list of most frequently used registers, pushing the link register on position seven. The static register allocation might benefit from this method as well.
Condition codes can even be optimized on a higher level: With the help of liveness information, it is possible to find out, what condition register write instruction (compare or ”.”-instruction) belongs to which condition register read instruction (conditional jump). The conversion of the i386 flags into the signed representation can then be omitted if the conditional jump is changed to do an unsigned evaluation.

6.2.8.6 Other Architectures

The topic of this thesis is recompilation from RISC to CISC. The concrete combination of PowerPC and i386 has been used. It would also be interesting to investigate other RISC/CISC combinations.
A different RISC architecture as the source architecture will probably not make a big difference. A different CISC architecture as the target should be more interesting. Practically the only other important CISC architecture that is still produced today is the Motorola M68K, although it is constantly losing market share to the ARM family. It can certainly be argued whether it makes sense to target the M68K with a recompiler, but the project would nevertheless be interesting, as the M68K is very different to its CISCy cousin i386: It has 16 registers, but none of them are general purpose: 8 of them can only be used for arithmetic and logic operations, and the other 8 can only be used for memory accesses and indexing.
It would also be interesting, but make a lot more practical sense, to change the target of the recompiler to the AMD64 extensions, which seems to be the future of the i386 architecture, now that both AMD and Intel support it in their latest CPUs. The main advantage of AMD64 are the eight additional registers, which makes 14 registers available for mapping PowerPC registers to. 85% of all register accesses target the most important 14 registers, which means that 73% of all instructions will need no memory access, which is a great improvement compared to 25%, when only 6 registers are available. It must be noted that these 73% can even be achieved using static register mapping—dynamic register allocation should even achieve percentages very close to 100% then, as in practice, few functions really need more than 14 registers.
Just like the i386 line is being migrated to 64 bit using AMD64, the desktop PowerPC world is shifting to 64 bit CPUs. Apple is using the 64 bit IBM PowerPC 970 (G5) in all iMac and PowerMac models today, and Mac OS X runs a great portion of the code in 64 bit mode. It would be very interesting to see how well the 64 bit PowerPC can be mapped to the AMD64.

6.2.9 Other Applications

The recompiler that has been developed has been designed for user mode PowerPC to i386 recompilation. It may be possible though to extend the design for system emulation, or even adapt the design for the implementation as a recompiler in hardware.

6.2.9.1 System Emulation

A recompiler that has to emulate a complete computer system needs a higher level of accuracy than a recompiler only doing user mode emulation.

Interrupts If a scheduler/timer interrupts occurs during a calculation in user mode for example, the operating system wants to save the contents of all registers to memory and possibly switch to another thread. When the original thread gets switched to later again, the operating system loads the contents of all registers from memory again and continues execution at the same point.

Interrupting execution at any point and saving the registers is impossible for a recompiler with basic block accuracy, because basic blocks are atomic and there is no state of the source CPU inside during the execution of a translated basic block. In particular, there is no way to reconstruct the contents of the source registers inside a basic block, unless register mapped registers are written to memory and the function that tests whether an interrupt has occurred is called after every instruction, which pretty much lowers the execution speed to the level of a threaded code recompiler.
Instead, registers could be written back and the interrupt check function could be called after every block of execution only, but this delays interrupts. Therefore, basic block recompilers typically are not cycle exact[3]. In most cases, this level of accuracy is sufficient.

Memory Emulation When doing user mode emulation, a lot of memory addresses are variable. The stack for example, can be located at an arbitrary address, and the application won't complain. A system emulator cannot just put the stack anywhere, but must always emulate the exact content that the emulated register defines for the stack pointer. As this address can point anywhere, it is generally not possible to put the emulated stack at the same position.
The same is true with most other addresses: The emulated operating system may make full use of the complete addressing space. So it is not possible for example to translate an instruction that accesses memory at the address which is stored in a register into the equivalent in the target language, because this address might not be available to the emulator as it might be occupied by the operating system. Instead, addresses have to be adjusted every time before they are used.
Furthermore, operating environments that use virtual memory are even more complicated to emulate, as the complete memory management unit of the system has to be emulated as well. For every memory access, the virtual address used by the application has to be converted into a physical address. If this is done in software, it typically takes a minimum of 50 cycles [55]. Doing it in hardware by configuring the host MMU to do the address translation can be very complex or even impossible for certain combinations.

Simplifications Link Register emulation may not be simplified by placing target addresses into the emulated link register as it has been done in this project, because the operating system will read and write to the link register and might even perform tests or arithmetic on it. The same is true for the condition register. Emulation needs to be a lot more exact and less simplifications are possible.

6.2.9.2 Hardware Implementation

The pass 1 instruction recompiler is very simple. In its easiest configuration, it only tests for a simplified meaning of a source instruction, decides what block of code to emit depending

[3]The Transmeta Crusoe [6], a VLIW CPU with an i386 personality, which recompiles i386 code into its native instruction set, is cycle exact although it translates blocks of code. This is achieved by a hardware based rollback functionality that can undo all operations of the last basic block. The block is then interpreted up to the instruction, after which the interrupt must be processed.

on whether the parameters are register mapped or memory mapped, and finally emits the block of code with the corresponding registers and memory addresses inserted.
Most of this design can be quite easily implemented in hardware as well. The input of the unit would be the instruction code. After extracting the opcode field(s), a table can be used to look up the parameter fields of this instruction. Another lookup in the register mapping table would decide whether the register operands are register or memory mapped. Yet another table, indexed by the opcode and the mapping type of the registers, can contain the blocks of i386 code and the information where to fill the registers and memory locations.
Detection of special meanings could be added as well, but this is more complicated. The basic block logic and cache however should be easier to implement.

6.3 SoftPear

> ”Most people have two reasons for doing anything — a good reason, and the real reason.” - *fortune-mod*

As the main purpose of this project for me was to develop the basis of a recompiler for the Open Source SoftPear project, which aims to make the Mac OS X operating system available on i386 hardware, this will not be the end of this recompiler project. Although ”ppc_to_i386” is not very usable yet, it is a solid base for future development.

Appendix A

Dispatcher Speed Test

Chapter 2 refers to the maximum speed of an interpreter loop or a recompiler loop, and the costs of the dispatcher. The following assembly program has been used to conduct these measurements:

```
.global main
main:
    pusha
    mov $constant, %esi
    mov $1200000000, %ebp
interpreter_loop:
    mov (%esi), %eax
    xor $4, %esi
    mov %eax, %ebx
    and $0x0000ffff, %ebx
    cmp $1, %ebx
    ja opcode_1_is_not_31
    mov %eax, %ebx
    and $0x0000ffff, %ebx
    jmp *dispatch_table(,%ebx,4)
target1:
    nop
target2:
    mov %eax, %ebx
    shr $21, %ebx
    and $31, %ebx
    mov %eax, %ecx
    shr $16, %ecx
    and $31, %ecx
    mov %eax, %edx
    shr $11, %edx
    and $31, %edx
    mov registers(,%ebx,4), %eax  // execute
```

```
    or  registers(,%edx,4), %eax
    mov %eax, registers(,%ecx,4)
    dec %ebp
    jne interpreter_loop
opcode_1_is_not_31:
    popa
    ret

.data
registers:
 .long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
 .long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

constant:
 .long 0
 .long 1
dispatch_table:
 .long target1
 .long target2
```

This loop basically does the same as the interpreter loop and therefore has the same speed, but it is simpler. Instead of adding 4 to the program counter every time, it just jumps between two fake instructions, which will be dispatched to target1 and target2. The jump will therefore not be predictable, just like in a real interpreter loop. The number of iterations has been set to the clock frequency of the CPU in Hz (1200 MHz in this case), so the time that the program needs to execute will be the number of cycles per iteration. As is, the program takes 28 cycles per iteration on a Duron 1200 MHz.
The program can be changed to measure the costs of the jump, by simply removing it. The measured costs of the program without the jump are 13 cycles, so the jump costs 15 cycles. When changing the ”xor” instruction to modify another register than ESI, the same instruction gets interpreted every time, which has the same costs as if the jump were not there, as the CPU can do branch prediction.
The costs of the dispatcher alone, or the costs of a recompiler loop can also be measured by removing the three execution instructions, respectively by replacing them with emitter code.

Appendix B

Statistical Data on Register Usage

This appendix contains the statistical data that has been used for the register usage frequency graphs in chapter 3.

Table B.1: Register Usage Frequency

r0	1494142	9,4%	r14	32745	0,2%	r28	388292	2,4%
r1	1546649	9,8%	r15	31815	0,2%	r29	580875	3,7%
r2	1079458	6,8%	r16	31130	0,2%	r30	752249	4,7%
r3	1810318	11,4%	r17	36628	0,2%	r31	670504	4,2%
r4	1137901	7,2%	r18	41429	0,3%	ctr	180346	1,1%
r5	617436	3,9%	r19	46185	0,3%	lr	979858	6,2%
r6	340545	2,1%	r20	50467	0,3%	cr0	629589	4,0%
r7	639833	4,0%	r21	59606	0,4%	cr1	21045	0,1%
r8	165877	1,0%	r22	72414	0,5%	cr2	2532	0,0%
r9	497445	3,1%	r23	85101	0,5%	cr3	4368	0,0%
r10	160761	1,0%	r24	113385	0,7%	cr4	19484	0,1%
r11	227454	1,4%	r25	143373	0,9%	cr5	1383	0,0%
r12	204925	1,3%	r26	200226	1,3%	cr6	22342	0,1%
r13	27948	0,2%	r27	268444	1,7%	cr7	435202	2,7%
						SUM	15851709	100,0%

Table B.2: Register Usage Frequency (Byte Accesses)

r0	44795	41,6%	r16	407	0,4%
r1	166	0,2%	r17	159	0,1%
r2	13441	12,5%	r18	435	0,4%
r3	7651	7,1%	r19	209	0,2%
r4	4783	4,4%	r20	264	0,2%
r5	4162	3,9%	r21	292	0,3%
r6	2829	2,6%	r22	413	0,4%
r7	2265	2,1%	r23	416	0,4%
r8	2355	2,2%	r24	567	0,5%
r9	3817	3,5%	r25	871	0,8%
r10	2497	2,3%	r26	1209	1,1%
r11	3320	3,1%	r27	1743	1,6%
r12	1727	1,6%	r28	1692	1,6%
r13	168	0,2%	r29	2385	2,2%
r14	221	0,2%	r30	1495	1,4%
r15	155	0,1%	r31	697	0,6%
			SUM	107606	100,0%

Bibliography

[1] Apple Computer, Inc.: *Technical Note PT39: The DR Emulator*, http://developer.apple.com/technotes/pt/pt_39.html

[2] Microsoft Corporation: *Microsoft Virtual PC for Mac*, http://www.microsoft.com/mac/products/virtualpc/virtualpc.aspx

[3] *VMware*, http://www.vmware.com

[4] Virtio Corporation Limited: *Virtio Virtual Platforms* http://www.virtio.com/

[5] Transitive Corporation: *QuickTransit* http://www.transitive.com/PDF%20Docs/QT_Overview_f.pdf

[6] James C. Dehnert et al.: *The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges*, http://citeseer.ist.psu.edu/dehnert03transmeta.html

[7] Anton Chernoff, Ray Hookway: *DIGITAL FX!32 — Running 32-Bit x86 Applications on Alpha NT*, http://citeseer.ist.psu.edu/462062.html

[8] Anton Chernoff et al., *FX!32 - A Profile-Directed Binary Translator*, http://citeseer.ist.psu.edu/chernoff98fx.html

[9] Leonid Baraz et al.: *IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems*, http://citeseer.ist.psu.edu/645329.html

[10] Sun Microsystems: *The Java HotSpot Virtual Machine, v1.4.1, d2 — A Technical Whitepaper*, http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf

[11] *Microsoft .NET* http://www.microsoft.com/net/

[12] *The Computer History Simulation Project*, http://simh.trailing-edge.com/

[13] *VICE*, http://www.viceteam.org/

[14] *UAE Amiga Emulator*, http://www.freiburg.linux.de/ uae/

[15] *PearPC - PowerPC Architecture Emulator*, http://pearpc.sourceforge.net/

[16] Fabrice Bellard: *QEMU Internals*, fabrice.bellard.free.fr/qemu/qemu-tech.html

[17] *Basilisk II JIT*, http://gwenole.beauchesne.online.fr/basilisk2/

[18] *Hatari*, http://hatari.sourceforge.net/

[19] *UltraHLE*, http://www.ultrahle.com/

[20] David Sharp: *TARMAC — A Dynamically Recompiling ARM Emulator* http://www.davidsharp.com/tarmac/tarmacreport.pdf

[21] Julian T. Brown: *ARMphetamine — A Dynamically Recompiling ARM Emulator*,

[22] *Wine*, http://www.winehq.com/

[23] *Darwine* http://www.opendarwin.org/projects/darwine/

[24] Cristina Cifuentes et al.: *The University of Queensland Binary Translator (UQBT) Framework*, http://experimentalstuff.sunlabs.com/Technologies/uqbt/uqbt.pdf

[25] *"dynarec" Group*, http://groups.yahoo.com/group/dynarec/

[26] *"staticrecompilers" Group*, http://groups.yahoo.com/group/staticrecompilers/

[27] Graham Toal: *An Emulator Writer's HOWTO for Static Binary Translation*, http://www.gtoal.com/sbt/

[28] *SoftPear PC/Mac Interoperability*, http://www.softpear.org/

[29] Michael Steil: *SoftPear Project Log*, http://www.softpear.org/project_log.html

[30] John L. Hennessy, David A. Patterson: *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann, 1995

[31] John L. Hennessy, David A. Patterson: *Computer Organization and Design*, 2nd ed. Morgan Kaufmann, 1997

[32] IBM Microelectronics: *PowerPC TMMicroprocessor Family: The Programming Environments for 32-Bit Microprocessors*, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2/$file/6xx_pem.pdf

[33] IBM Microelectronics: *Developing PowerPC Embedded Application Binary Interface (EABI) Compliant Programs* http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569970071B0D6/$file/eabi_app.pdf

[34] Apple Inc.: *Mach-O Runtime Architecture for Mac OS X 10.3*, http://developer.apple.com/documentation/DeveloperTools/Conceptual/MachORuntime/MachORuntime.pdf

[35] Intel Corporation: *Intel 80386 Programmer's Reference Manual*, http://www.fh-zwickau.de/doc/prmo/docs/386intel.txt

[36] Intel Corporation: *Intel Architecture Software Developer?s Manual Volume 2: Instruction Set Reference*, http://developer.intel.com/design/pentium/manuals/24319101.pdf

[37] Advanced Micro Devices, Inc.: *AMD Athlon TM Processor x86 Code Optimization Guide*, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf

[38] Intel Corporation: *IA-32 Intel Architecture — Optimization Reference Manual*, ftp://download.intel.com/design/Pentium4/manuals/24896611.pdf

[39] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers — Principles, Techniques, and Tools*, Addison-Wesley, 1986

[40] Arjan Tijms: *Binary translation — Classification of emulators*, http://www.liacs.nl/~atijms/bintrans.pdf

[41] Joshua H. Shaffer: *A Performance Evaluation of Operating System Emulators*, http://www.eg.bucknell.edu/~perrone/students/JoshShaffer.pdf

[42] Jochen Liedtke et al.: *An Unconventional Proposal: Using the x86 Architecture As The Ubiquitous Virtual Standard Architecture*, http://i30www.ira.uka.de/research/documents/l4ka/ubiquitous-vs-arch.pdf

[43] Cristina Cifuentes, Vishv Malhotra: *Binary Translation: Static, Dynamic, Retargetable?*, http://citeseer.ist.psu.edu/cifuentes96binary.html

[44] Mark Probst: *Fast Machine-Adaptable Dynamic Binary Translation*, http://www.complang.tuwien.ac.at/schani/papers/bintrans.ps.gz

[45] Georg Acher: *JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler fr eingebettete Anwendungen*, http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2003/acher.pdf

[46] Leonidas Fegaras: *Design and Construction of Compilers*, http://lambda.uta.edu/cse5317/notes.pdf

[47] E Christopher Lewis: *Introduction to Data-flow Analysis*, http://www.cis.upenn.edu/~cis570/slides/lecture03.pdf

[48] Anton Ertl: *Threaded Code*, http://www.complang.tuwien.ac.at/forth/threaded-code.html

[49] James R. Bell: *Threaded Code*, CACM, 1973, 16, 6, pp 370-372

[50] Stealth, Halvar Flake, Scut: *Spass mit Codeflow Analyse - neuer Schwung für Malware*, http://www.ccc.de/congress/2002/fahrplan/event/392.de.html

[51] Jon Stokes: *PowerPC on Apple: An Architectural History, Part I*, http://arstechnica.com/cpu/004/ppc-1/ppc-1-1.html

[52] Jon Stokes: *RISC vs. CISC: the Post-RISC Era — A historical approach to the debate*, http://arstechnica.com/cpu/4q99/risc-cisc/rvc-6.html

[53] Eric Traut: *Building the Virtual PC*, http://www.byte.com/art/9711/sec4/art4.htm

[54] Tom Thompson: *Building the Better Virtual CPU — Two different designs achieved the same goal: a faster 680x0 emulator for the Mac*, http://www.byte.com/art/9508/sec13/art1.htm

[55] Sebastian Biallas: *sepp_CPU_chat*, http://www.kelley.ca/pearpc/wiki/index.php/sepp_CPU_chat

[56] Fabrice Bellard: *QEMU Benchmarks*, http://fabrice.bellard.free.fr/qemu/benchmarks.html

[57] Neil Bradley: *Ms Pacman lives!*, Yahoo Groups posting, http://groups.yahoo.com/group/staticrecompilers/message/287

[58] Lukas Stehlik: *AmiGOD 2 — Benchmark*, http://www.volny.cz/luky-amiga/Benchmarks.html

[59] Raymond Chen: *Why does the x86 have so few registers?*, http://blogs.msdn.com/oldnewthing/archive/2004/01/05/47685.aspx

[60] Crystal Chen, Greg Novick, Kirk Shimano: *RISC Architecture*, http://cse.stanford.edu/class/sophomore-college/projects-00/risc/

[61] Bart Trzynadlowski: *68000 Instruction Usage Statistics*, http://www.trzy.org/files/68kstat.txt

[62] Wikipedia, the free encyclopedia: *PowerPC*, http://en.wikipedia.org/wiki/PowerPC, http://de.wikipedia.org/wiki/PowerPC

[63] John Mashey: *Re: RISC vs CISC (very long)*, comp.arch newsgroup posting, message ID <2419@spim.mips.COM>, http://groups.google.com/groups?selm=2419%40spim.mips.COM&output=gplain

[64] *Geek.com — The Online Technology Resource!*, http://www.geek.com

[65] Gwenole Beauchesne: *Re: [dynarec] softpear*, Yahoo Groups posting, http://groups.yahoo.com/group/dynarec/message/518

[66] *Melissa Mears*, http://www.xbox-linux.org/User:Melissa_Mears

[67] Julian T. Brown, *ARMphetamine "oldinfo"*, http://armphetamine.sourceforge.net/oldinfo.html

[68] *AMD Opteron Server Processor*, http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796,00.html

[69] *VIA C3 Processor*, http://www.via.com.tw/en/viac3/c3.jsp