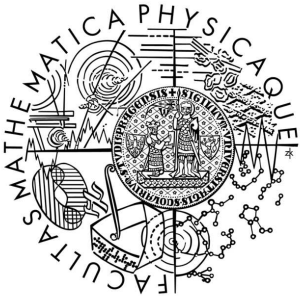
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE
MASTER THESIS



Alexandr Kára

Optimalizace binárního kódu pro Intel IA-32

Optimization of binary machine code on Intel IA-32

Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Mikuláš Patočka
Studijní program: Informatika, Softwarové systémy

I would like to thank Mikuláš for great encouragement in writing this thesis and for valuable suggestions and corrections – both to the text and to the program. Many others supported me throughout the work. A big thanks goes to Linda for a lot of patience and support. Homer and Nastěnka helped me with the last time corrections.

I am also grateful of all the work of thousands of volunteers who created invaluable sources of information, such as Wikipedia or the abundance of other online materials, and to the programmers who wrote the tools and libraries I worked with.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 14.12.2006

Alexandr Kára

Název práce: Optimalizace binárního kódu pro Intel IA-32

Autor: Alexandr Kára

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Mikuláš Patočka

e-mail vedoucího: mikulas@artax.karlin.mff.cuni.cz

Abstrakt: Překladače dnes používají mnoho technik pro zrychlení přeloženého programu a dalších zlepšení se dosahuje poměrně komplikovanými způsoby. Cílem této práce je vyzkoušet na základě profilovacích dat optimalizovat již přeložený a slinkovaný program. Tento přístup má několik potenciálních výhod. Umístění jednotlivých částí kódu a jejich relativní vzdálenosti jsou již známy a díky tomu lze zvyšovat lokalitu a linearitu kódu, které v důsledku zvyšují efektivitu cache pamětí. Další výhodou je možnost pracovat s programy, u kterých není k dispozici zdrojový kód. Takový program je pak možné bez dalších informací upravit pro efektivnější práci na jiném typu procesoru (Intel Core nebo AMD Athlon64).

Klíčová slova: optimalizace "binární optimalizace" "binární kód" profilování

Title: Optimization of binary machine code on Intel IA-32

Author: Alexandr Kára

Department: Katedra softwarového inženýrství

Supervisor: Mgr. Mikuláš Patočka

Supervisor's e-mail: mikulas@artax.karlin.mff.cuni.cz

Abstract: Compilers use currently many techniques for optimizing compiled code. To further enhance their efficiency, complicated methods are required. The goal of this thesis is to try to optimize a compiled and linked program, using profiling data. This binary optimization approach has several potential advantages. It is only after the linker stage that the exact positions and especially relative distances between individual chunks of code are known - and improved code locality and linearity can significantly influence the efficiency of processor caching. Another advantage is the ability to work with programs without access to source code. These programs can be without any additional information modified to work faster on a particular processor family (such as Intel Core or AMD Athlon64).

Keywords: profile-driven install-time optimization "binary code"

Table of contents

1 Introduction

In this thesis, I propose a *IA-32 Binary Optimizer* framework for optimization of binary executable programs on the IA-32 architecture. It is a general optimization and profiling framework for Linux working with ELF binary files. It reads, analyses and modifies binary code without a need for sources, special symbols or debugging information. In the current implementation, the main performance can be expected for programs where instruction cache performance is a bottleneck. The thesis is accompanied by source code of the framework programs that perform the optimization.

The text is organized as follows: The rest of *Chapter 1* gives a short introduction to various optimization approaches and an overview of the framework, *Chapter 2* discusses processor microarchitecture and presents possibilities for optimizations, *Chapter 3* briefly introduces the ELF file format used for the optimization and as library to work with it, *Chapter 4* is a detailed description of the architecture of the framework and documentation of choices made during development. Experimental results and benchmarks are presented in *Chapter 5*, other optimization techniques, that could be implemented in future, and room for improvement are summarized in *Chapters 6 and 7*.

Overview of the program source code and important data structures used is presented in *Appendix A*. Usage, installation, configuration and command-line options of various tools in the framework is set out in *Appendix B*. For a step-by-step example of an optimization session, visit *Appendix C*.

1.1 Code optimization on native and intermediate code

Many approaches are used to produce optimized binary code. For languages which traditionally compile to native code directly, such as C++, optimization is done in the compiler in most cases. Languages that use a non-native intermediate code or bytecode, such as Java or C♯, compile to native code by a JIT (Just-In-Time) compiler just before execution. This has a number of potential benefits: it can adapt to the particular machine it is running on, it can be compiled on the fly and parts may be recompiled when necessary or when a change in usage pattern is detected.

For languages that don't have an intermediate code, all decisions are made during compilation. In many cases, the target machine is not known and the usage pattern is not taken into account. It is possible to use profiling data from previous executions of the program to enhance optimization in compiler, but this approach is rather cumbersome, requires modifications to the build process, and is not used very often.

Even if profiling data is used during compilation, the resulting application is often designed to run on many different microarchitectures, such as various generations of Intel or AMD processors. This forces the compiler to use the lowest common denominator of all supported processors which often have very different scheduling requirements or support particular instructions or instruction combinations more

efficiently than other microarchitectures. Even if code was compiled in several versions for different processors, it may become outdated when a new processor line is introduced. Specially optimized code for one processor often runs more slowly on newer generations of the processor than code optimized less aggressively.

In many situations, neither the source code nor a version compiled for a current processor is available.

Another disadvantage of compilation to native code is that the compiler usually never sees the whole program at once, because the compilation is performed on compilation units, usually one source file at a time. Optimizations are often performed on even smaller scale, one function or even one basic block at a time. The compiler must therefore make rather conservative assumptions and may miss many optimization opportunities.

Interprocedural optimizations may work on a whole compilation unit at a time, but time-critical hot-spots[1] often cross boundaries of compilation units. For tight loops, all code might be put in one file to be compiled at once. This, however, might not be always possible or desirable because of manageability or coding style. Compilers also don't know about final addresses of the code and have little opportunity to prevent cache conflicts between two parts in different compilation units.

Applications can be loosely classified into two categories: loop-intensive programs and call-intensive programs. In loop-intensive programs, the important loops are usually inside one function or at least in one compilation unit. This type of program can be optimized fairly well by a conventional compiler. In call-intensive programs, on the other hand, the most important loops may span multiple functions which are often distributed among several compilation units. Control flow is complicated and it is difficult to recognize correctly the most frequently used parts. Optimizations for such programs must therefore be interprocedural and sometimes covering the whole program, not just a single compilation unit. For this type of programs, a post-pass or install-time optimizer, which uses profile data and optimizes the final binary code, can provide a useful supplement to a traditional compiler, in cases when performance is critical.

After a program has been compiled, there is very little structural information left and it is therefore difficult to analyse and modify such files. It is sometimes impossible to predict a possible control flow or distinguish data from references to code. This can happen when using indirect calls through function tables, virtual method tables, passing a function callback reference to another function and in other cases. If the code at the original position changes, the addresses in the data, which reference it, should be updated – but only if they really did represent the address, not if the value just happened to be the same.

One approach to solving this problem is to require that the optimized programs have some additional meta-information embedded. Other possibilities include leaving parts of the program on the original addresses or attempt to analyse the program deeper, which is often not possible.

1. A hot-spot is a place in the code, which is most frequently executed.

There have also been attempts to use run-time optimizations for native code with special hardware support in the processor for profiling [1], but as far as I know, no such hardware has ever been constructed. Research is now also more focused on optimizations for JIT compilers, even if many speed-sensitive applications are still being written in the C/C++ or similar languages.

1.2 Overview of the IA-32 Binary Optimizer system

In this thesis, I propose a framework for install-time[2] optimization for the Linux operating system called the *IA-32 Binary Optimizer*. It processes binary executable programs and generates new executable files that should be better tuned for the particular processor it runs on.[3]

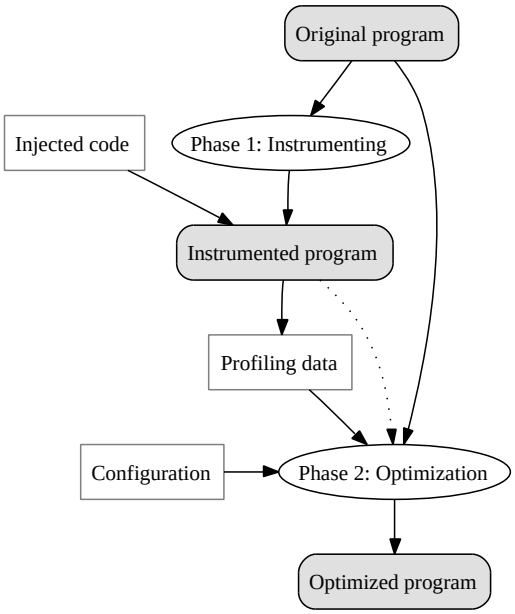


Figure 1. Overview of the optimization framework

It operates in two stages. First, the binary file is instrumented to generate profiling data during execution. Special small pieces of code are inserted at the beginning and at the end of most basic blocks[4] and an initialization code is injected before the program entry. These bits of code record how many times the control has passed through the particular block, how many times a conditional branch has been taken and possibly other information.

When the instrumented binary is run, the profiling data is gathered and then stored to a file under a special directory. The profiling data size is constant and doesn't grow with the execution time or multiple executions.

2. Also called post-link, post-pass or late-code optimization in literature.

3. Currently, only AMD Athlon XP and to some extend Intel Pentium-M and Core/Core2 processor families are supported, but support for other processors should be straightforward.

4. Basic block is a linear stretch of instructions with control flow inflow only at the beginning and exit only at the end, with no jumps inside except possibly at the end. In some definitions, calls are also not allowed.

Some profiling data could be obtained using statistical sampling-based profiling, which has a lower overhead and doesn't need to modify the executable. It would be useful for hot-cold optimizations, but it would be difficult to obtain more detailed information about the execution behaviour. Sampling-based methods have also problems with high correlation to timer interrupts or other event sources, but this has supposedly been solved in some applications [2].

The profiling data gathered is used in the next stage of the framework, the optimizer. It reads both the original file and the instrumented file to correctly interpret the profiling data. Before optimization begins, various analysis of the program are made. A control flow graph, annotated with pass counts, and a dominator tree of the basic blocks are constructed. Stack pointer value is analysed and live[5] registers and stack areas are identified.

The actual optimization is performed by different optimization plugins. There are four plugins actually implemented, but adding new plugins shouldn't be too difficult.

The first one – *CacheUnalias* plugin – aims at eliminating cache aliasing problems by trying to make the most frequently executed paths straight, group code that executes together and to a smaller extend also separate *hot* parts[6] of the program from the *cold* parts.[7] This can help with processor code cache utilization as well as with paging, requiring less pages to be resident.

The second plugin, called *BranchAlign*, aligns blocks that are targets of frequent jumps and lay at the start of a *hot* area to a multiple of a number, usually 8 or 16 on AMD architectures. This helps some processors to process loops more effectively.

The *AthlonBTB* plugin improves branch target buffer performance of the `RET` instruction on Athlon processors by changing its encoding to be two bytes long.

The last implemented, the *HotColdSeparate* plugin, separates *hot* and *cold* code – but on a function granularity, unlike the *CacheUnalias* plugin.

After all configured plugins perform their work, the optimized code is written to a new file. Because some code references cannot be identified with certainty, some code from the unoptimized binary needs to remain in place. At addresses in the original code that are selected as likely targets of a jump, a *trampoline* is placed. It is a jump to the corresponding location in the new code – optimized or instrumented.

An implementation of SSA analysis is under way and when finished, should open possibilities for other optimization techniques. For example eliminating superfluous checks for return values from functions that only return a constant or checking a value that has already been checked before. Also dead code elimination or register reallocation may be done more efficiently with the SSA form available. Such optimizations are already well supported in current compilers, but it will be possible to perform them across compilation units.

5. A variable (or a register or memory place) is said to be live at a certain point in the program, if its value may potentially be used later.

6. A hot part of a code is an area that is executed often.

7. By analogy, a cold part is an area that is not executed so often.

1.3 Potential uses and limits of the optimization

1.3.1 Limitations of a binary optimization framework

As was said before, a binary optimization framework is limited in the optimization capabilities by the lower amount of information it has about the optimized program. It is not meant to replace a compiler backend, but to provide additional possibilities in optimization, such as an optimization targeted for a particular processor.

If the optimized binary doesn't have relocation information, all code that could possibly be executed by an indirect jump which cannot be predicted with certainty, must be left at its original address. Such programs will therefore necessarily grow during optimization and jumping into the original code parts that were left at their original addresses can impair the optimization done on the file.

Of course, compilers could generate relocation information and other data that would help to process the binary on the target machine, but why not then compile to an intermediate code and do the work before running the application first?

One of the problems of systems working with intermediate code is the need to have a runtime framework support to compile and run the program (while binary code works "out-of-the-box"). In future, it is probable that such support will become a standard part of every computer and there will be no need to distribute native code any more. Compilation of intermediate code on the final machine has probably the potential to outperform native code unless compiled for the specific processor it later runs on. Distributing applications in the source code format can have the same possibilities, but there are problems with long build times, virtual non-existence of standard ways to specify compiler options[8] or dependencies on other applications or libraries.[9] A traditional compiler is also much more heavyweight than a JIT compiler and many vendors are not willing to provide source code of their applications. Until intermediate code is widely supported, binary programs are the most convenient way to distribute applications.

A success full binary optimization framework must therefore be able to operate on programs that are already heavily optimized for a particular or generic processor and ready to run. Binaries can then be shipped without the need of an optimization framework and only users that require the optimization can process the binary for themselves.

A special problem for analysing binaries is hand-written assembler code that doesn't

8. Compiler and other tools options are often incompatible between different versions of the same compiler. The Autoconf/Automake/Libtool is one approach to overcome the problems, CMake is another, but both have their limitations.

9. The Gentoo Linux distribution uses this approach and compiles all packages from source code on the target machine. It requires all packages to be "Gentoo-enabled" and have a standard build "recipe", which also specifies all dependencies.

respect certain paradigms such as that a `CALL` instruction is used to call a function or that functions may not overlap. I tried to deal with such programs as much as possible by not relying on a particular structure of the code, but if a code is very unusual in some aspects, it may be refused or poorly optimized. Self-modifying code will likely not work as expected, either.

Although in theory it is possible to work with programs that use exceptions, it is currently not supported.

The exception handling ABI is not unified – the Gcc compiler uses its special variant of DWARF2 debugging format for stack unwinding, but can also use *setjmp()/longjmp()* for exceptions. Other compilers may use different exception handling schemes. The work on DWARF exception support is in progress, though.

1.3.2 Other uses of the framework

Optimizer plugins don't need to modify the code – it is not hard to imagine plugins that check correctness or security aspects of a closed-source application. Plugins may also insert some code into arbitrary places, such as before system calls, to check or modify their parameters, or before an indirect jump to check its target.

The analyser could be used to check assertions about program behaviour which are hard to verify on source code level because of various preprocessor modifications and compilation options. It may also be used as a profiler that doesn't need a special build procedure. With some modifications, it might be used as a JIT compiler, which would optimize (or translate) code on the fly. This concept could prove useful in long numeric calculations, where hot spots and jump patterns may depend on the nature of data and change during a calculation.

The framework is written to impose as little requirements on the optimized program as possible. It can work without any symbols or relocations present in the program. One of the purposes of this thesis was actually to demonstrate that even a stripped binary with no relocation or debug information[10] may be further processed. Functions are therefore not that important and may be freely merged, joined or partially/fully inlined.

The system can be extended to use the x86-64 (AMD-64) instruction set, and some work has already been done on that.

10. If there is some debug information in the DWARF format, the framework tries to adjust it for the final optimized program. Currently, it works with line numbers to improve experience with debugging optimized or instrumented files.

1.4 Related work

1.4.1 IMPACT

The *Impact* binary reoptimization framework from the University of Illinois [3][4][5] is a system for optimizing Windows NT programs on the IA-32 architecture. It aims at optimizing programs for the particular machine the framework is running on, without requiring source code for the application.

The program is first instrumented by adding counters to the code. When running the instrumented version, profiling data is gathered and used later for optimization.

IMPACT works by parsing x86 instructions first and translating them to an intermediate code called Mcode. This code is not tied to x86 architecture, but has a specific one-to-one mapping to x86 target machine instructions. It also contains control flow and data flow information. Both instrumentation and optimization is performed on Mcode which is then translated back to machine code and written to a new program.

Several optimization techniques are used, most notably code reorganization to separate *hot* and *cold* parts of a program and instruction rescheduling.

Code is parsed conservatively, following only possible branches to avoid accidentally parsing non-instruction data embedded in a code section. Special care is taken for indirect jumps through jump tables. Displacement in the instruction is used to locate the range of the jump table and heuristics are used to find the bounds of such tables.

The optimized and instrumented code is written to the same address range as the original code, but addresses of particular functions or blocks may change. All references to code addresses must therefore be updated to new locations of the corresponding block in the optimized code. IMPACT uses a special relocation table (on Windows, it is called Base Relocation Table, BRT). This table is used to find all absolute references to code and to identify constants that should be treated as addresses and which should be treated as data. If a constant doesn't fall into the address space of the program, it can be safely assumed that it doesn't represent code address, but not the other way round. If a constant, which is not a pointer, was treated as such and updated to point at the new location of the transformed code or moved data, it could result in a program malfunction.

Unfortunately, Visual Studio .NET by default doesn't generate BRT table any more for programs (as described in [6]), so *IMPACT* probably couldn't be easily used on a more recent system.

The instrumented binary is executed in debug mode as a child process of a controlling process called monitor. The monitor gathers and interprets all profiling data and can take appropriate actions upon notifications of events such as loading of a dynamic library or process exit. Just before the controlled process terminates, the monitor reads its counters and other profiling data and stores them to a file.

1.4.2 Spike

The *Spike* Optimization Environment [8] from Digital (now part of Compaq) is a system for performing profile-feedback optimizations of code for Windows NT on the Digital Alpha processors. It modifies the code layout to improve instruction cache behavior, and also uses hot-cold optimization and register allocation. It can reach up to 33% improvement of execution time.

In the first step, it instruments the binary to be optimized. To perform its job, it requires that the binary contains relocation information, so that all data that represent addresses are understood and can be modified when addresses change during instrumentation or optimization phases.

Then, the instrumented application is run by the user to collect profiling data, which will be stored in a special system database. The optimizer then uses this information to optimize the binary.

Three optimizations are used: code layout optimization, hot-cold optimization and register allocation. The code layout optimization tries to reduce the number of code cache used and also the number of active VM pages needed. It works on each routine separately and rearranges basic blocks so that the most frequently executed path is made straight, using a simple greedy algorithm. The hot-cold optimization separates *hot* and *cold* parts of each routine and tries to place *hot* code from all routines next to each other and place the cold code at the end of the process image. Hot code from routines that frequently call each other is placed closer together.

This reorganization should improve the code cache behavior by making code run straight and reducing the probability that addresses of two routines frequently calling each other clash in the code cache. The instruction fetching should also be improved, because branches are mostly not taken and instructions are executed sequentially. It should also reduce the number of active, frequently used VM pages.

The last optimization is a register allocation optimization which tries to improve poor register allocation done by the compiler.

The *Spike* optimizer has to find all references to code and data to be able to modify them when the code or data is moved during optimization or instrumentation. Branch targets for PC relative branches are simple to recognize, and relocations are used for addresses in data or literals in instruction encoding.

It uses the original instruction encoding along with a small annotation as a base for its internal representation. On top of that, it constructs basic blocks, control-flow graph, call graph and other data structures.

1.4.3 Digital FX!32

Digital *FX!32* [10] is a commercially used package for running Windows NT applications compiled for the IA-32 architecture on Digital Alpha processors. It combines an x86 processor emulator with binary translation. When an application is first executed, *FX!32* uses a x86 processor emulator which also captures execution profile data used later by a binary code translator. The code translator converts x86 code into native Alpha code.

The emulator records information such as addresses of `CALL` instruction targets, (source address, target address) for indirect control transfer instructions and addresses of instructions that make unaligned access to memory. The collected data is further processed when the application terminates or the library is unloaded.

The translator uses this information to generate native code for routines for which a profile exists. The emulator looks up the target address every time it encounters a `CALL` instruction and calls native code if it exists. After the application terminates, a retranslator is triggered which translates code to native Alpha instructions every time a new part of the program is executed in the emulator. This results in gradually expanding the native code after the application is executed several times.

The emulator and the translated code both use so called "jackets" on every system call entry that translate the contents of the top of the stack from x86 to Alpha conventions. These jackets are highly dependent on particular Windows version, but the *FX!32* system doesn't require any modification to the underlying operating system.

1.4.4 Sun Studio Binary Code Optimizer

The Sun Studio *Binary Code Optimizer* [13] is a new profile-guided static optimization system for SPARC processors. It is a part of Sun Studio 11 release.

It works with binary programs that contain additional information in a separate `.annotation` section produced when compiled with a special compilation option. The information in this section include location of executable code structures like functions, switch tables and data flow information. This additional data increase the binary by about 5% on average but doesn't incur any run-time performance cost.

A binary, that is compiled with the necessary information, can be instrumented. When it is running, it collects profiling data. This data is later used in the optimization stage. The optimized binary retains the original code and places the optimized code in a new segment, which makes is about 50% to 80% bigger that the original version.

Reports suggest performance increase of up to 10%, especially if the binary hasn't been compiled with profile feedback or with a feedback different from production use.

1.4.5 Charm

Charm [14] is a commercial static binary code optimizer for StrongARM, XScale and other embedded processors. It uses profile data generated using a tool from the *Pin* framework [15] (see below) to optimize the binary using dead-code elimination, limited inlining and code layout optimizations.

It works with a limited set of compilers at specific versions, because it counts on matching specific code patterns in the executable to recognize various control flow structures. The stated run-time performance improvements can reach over 10%.

1.4.6 Morph

Morph [2] is a late-code profile-driven program optimization suite for the Digital Alpha processor on Digital UNIX system which works with already compiled and linked programs. It was created with three main key points in mind: it should optimize on the target machine to adapt to the end user's usage pattern and potentially target architecture, it shouldn't require source code and it should be transparent to the user.

For profile data gathering, it uses a statistical sampling-based method with a low runtime overhead but lower accuracy than profiling by instrumented code. For sampling, it requires a modification to the operating system by a special driver.

It doesn't require source code of the program to be optimized, but it requires a special representation of compiler intermediate form of that program. This is generated by a modified version of a SUIF compiler from Stanford University.

The optimizations performed by Morph include procedure layout to improve cache behavior, basic block ordering and improving branch prediction and instruction locality.

1.4.7 Etch

Etch [17] is an application performance evaluation and optimization system for Windows NT applications running on x86 systems. It works in three phases: first, the binary is instrumented to collect profiling data. Then, after some data is gathered, the program is optimized by reordering basic blocks so that the most frequently executed blocks are placed together. Other uses of Etch are possible, because the instrumentation framework can be given arbitrary instructions to insert into the binary, so any type of profiling can be done.

Etch is reported to achieve from under 5% to over 60% of instruction cache (code cache) miss reduction and 0% to over 10% of general performance improvement for test programs.

1.4.8 Valgrind

Valgrind [18] is a suite of free software tools for debugging and profiling Linux programs. Similarly to *FX!32* or *Pin*, it uses dynamic code translation to emulate an x86 environment, but mostly on a x86 machine.[11]

When used to debug programs, it loads itself as a shared library with them, seizes control and returns to initialize the other libraries and start the program – but not on the real CPU, only on a "synthetic", emulated processor.

11. There is support for other architectures than x86, but not for executing x86 code, because the profiled program has to use the same architecture as the Valgrind program that it is running in.

Portions of code are dynamically translated to an intermediate code called *UCode*. This code is architecture neutral, but the same instruction set is used for input and output. Debugging and profiling code are included into the *UCode* representation, it is optimized and the result is translated back into native code. These translated chunks of code are cached in a hash table and can be reused when the same code is executed again.

Valgrind runs application programs in a very similar environment that they would be executed in without it and does not need any modification of the host operating system or a kernel driver.

1.4.9 Pin

Pin [15] is dynamic instrumentation program provided free-of-charge by Intel. It supports a wide range of Intel processors on the Linux operating system. It does not optimize the code, but allows arbitrary code to be injected at various places in the executable process while running, without modifications to file on the disk.

Similar to *Valgrind* [18] (described above), it fetches parts of code, adds required instrumentation and translates them back to executable code using a JIT compiler. It also modifies the code slightly to ensure that the Pin runtime regains control after the translated part of code has finished executing. Although, it may have been originally inspired by *Valgrind*, it is now much more efficient in register allocations and other optimizations and is reported to be about three times faster than *Valgrind*.

The *Pin* framework is extensible and used by other tools, such as *Charm* (see above).

1.4.10 Qemu

Qemu [19] is a free x86, x86-64, PPC, ARM, Sparc or MIPS processor emulator, which uses binary translation to execute code on the host system. It uses a code table for all guest processor instructions, where each entry is a C code compiled for the guest system. It then concatenates entries from the code table to form a native representation of the basic block. It performs some optimizations on the resulting code, for example it tries to chain basic blocks where the control transfer is known at translation time, so that the code doesn't pass control to *Qemu* runtime too often.

Qemu can run both application programs and a complete operating system and doesn't need any modification of the host operating system nor a kernel driver.

1.4.11 aiPop

The *aiPop* optimizer suite [20] is a commercially available tool for post-link code size reduction for C16x/ST10, HC08 and ARM processors. It works on assembler sources or object files and employs several techniques to compact the code – such as functional abstraction (reverse inlining), tail merging, interprocedural optimizations using data-flow information, loop invariant code motion, peephole optimizations and other. The exact method of operation is not known, but hints are that it performs pattern matching on a pseudocode generated with other annotations such as control flow and data flow graph information.

1.4.12 Comparison with other binary code systems

The related systems for processing binary code described here can be divided into three main categories.

Optimizations systems are represented by *IMPACT* and *Etch* for the x86 platform, *Spike* and *Morph* for the Digital Alpha platform, Sun *Binary Optimizer* on the Sparc platform and *aiPop* and *Charm* for embedded processors. They differ in what type of input they require. *Morph* processes special compiler intermediate form, others process binary programs, but Sun *Binary Optimizer* requires special additional information embedded.

Emulators and binary translation frameworks are another group, represented by Digital *FX!32*, which translates x86 applications to run on Digital Alpha processors as normal processes on the Windows NT Alpha systems, and *Qemu*, which creates a whole virtual machine for emulation and is therefore operating system agnostic. Such systems can make deeper changes of how the code is executed, because speed is not usually as critical as compatibility. They can, for example, compile small parts on the fly into a translated code pool and throwing out code from the pool if space is needed.

The last group are programs that instrument the binary to gain some information about it. This category is represented by *Valgrind*, which is a debugger, *Pin*, a general instrumentation framework, and parts of *IMPACT* and *Morph* which provide profiling data. *Valgrind* translates the code on the fly and adds various checks for uninitialized data, memory operation errors and similar, the other two programs are parts of an optimization framework and translate a whole program at once. *Morph* is an exception in the programs described here – in that its input isn't a binary program but a compiler intermediate code.

The *IA-32 Binary Optimizer* framework described in this thesis has two parts, one is an optimizer and the other is a profiler. It operates on unmodified binary programs, which makes it similar to the *IMPACT* framework, *Charm* or Sun *Binary Optimizer*.

It differs, however, in several aspects. It is, as far as I know, the first binary optimizer for the Linux operating system on the x86 platform. It can also, unlike all other optimizers described here, work with standard, stripped binaries without requiring any additional information or assertions about compiler versions.

The Sun *Binary Optimizer* is very similar, but requires a special format of the executable program, the *Impact* optimizer framework uses a special relocation table (on Windows, it is called Base Relocation Table, BRT) for distinguishing addresses in code from data that accidentally also fall in the address space of code. This table has been produced by Windows compilers but now it is not included by default. *Charm* requires special version of the GNU Gcc compiler to work.

The *IA-32 Binary Optimizer* cannot count on relocations or additional data and must therefore make conservative assumptions about the code. In certain situations, it must leave the code in the original location or provide a trampoline jump to the new location. This can significantly increase the resulting binary size.

Similar to *Spike* it uses the original instruction encoding as a basis for its internal representation. On one hand, it limits the portability of the system, on the other hand, it is convenient for optimizers to know the exact instructions they work with. If another instruction encoding was used, and translated back to binary code, the optimizer could inadvertently clash with compiler's instruction selection and hurt the performance.

Spike and *FX!32* use a special agent called Transparent Application Substitution, which dynamically substitutes either the instrumented or optimized version of the application (if they are available) each time the original application is executed. While it is a convenient feature, it wasn't considered important enough to offset the problems in designing such system and justifying modifications to the underlying operating system.

2 Processor architectures

Processors employ various techniques to improve performance without increasing operating frequency too much. New generations of microprocessors often use new microarchitectures, which offer better performance with constructs that were slower in older models and vice-versa. It is therefore important to study the differences in order that we will be able to optimize code for a specific processor.

I will describe several families of microprocessors with focus on their differences and bottlenecks: Pentium/Pentium MMX, Pentium Pro/II/III, Pentium 4, Pentium M/Core/Core2, and AMD Athlon64. Even though the Pentium and Pentium Pro architectures are not used much today, there is still a considerable amount of code in use, which is optimized for these processors.

More information on processor architecture can be found in Intel manuals [21] [22], AMD Optimization Guide [23], overview of history of Pentium processors [24] and research papers by Agner Fog [25] [26]. Most of the low-level optimization tips come from [25] and [27].

2.1 Execution pipeline

To achieve higher throughput, processors use pipelining, they execute operations in several stages. When an operation is finished with one stage and advances to the next stage, operations behind it can enter the vacant stage. Generally, this can happen every clock cycle. The number of cycles between start of decoding of an instruction to the point where the results of the instruction are written is called *latency*, and generally increases as the pipeline gets longer. But as the number of instructions in flight[12] can increase, the total throughput is not affected as long as the pipeline works smoothly.

The stages can generally be split into branch prediction, fetch and decoding, register allocation and renaming, reordering and scheduling, execution, write-back and retirement. Some older architectures, such as Pentium 1, lack some of the stages.

Intel and AMD pipelines differ in many aspects but share the basic design.

2.1.1 Instruction fetch and decoding

The instruction data is first read into a decode buffer where the various fields of the instruction, such as prefixes or addressing modes, are decoded. The location from where to read and decode the data is often determined by a branch prediction mechanism, which tries to identify where will the code continue. Decoding is a complicated problem with the x86 instruction set, where instructions can have lengths from 1 up to 15, so there is usually a separate stage for instruction length decoding, which only determines where instructions start and how long they are. The fetch/decode unit can in many processors read 16 bytes and decode up to three instructions per clock cycle. The decoding stage is, however, often a bottleneck.

12. In-flight instructions/microoperations are operations that are being executed in one or another stage of the pipeline, at the same time. On some processors, there can be as many operations in flight as there are stages in the pipeline.

2.1.2 Microoperations

After decoding, instructions are usually split into one or multiple microoperations, which allow the execution core of the processor to use a RISC-like architecture. These microoperations[13] are simple enough to be executed directly, often in one cycle, without need of microcode or similar measures. Simple instructions are often translated into just one microoperation, but more complicated and less often used ones can translate to many microps.

There can usually be several microoperations in every stage of the pipeline, allowing a throughput of up to three or four microoperations per clock cycle.

Different classes of instructions generate a different number of microoperations. The simplest operations that only work on registers are often represented by one microoperation. Memory stores or *read/modify* instructions which read a memory operand and a register operand, perform some arithmetic or logic operation and modify the register and possibly flags, use two microoperations. Another class of instructions is *read/modify/write* instructions which read an operand from memory, perform an operation, possibly with a register, and write the result back to memory. These instructions usually translate to more microoperations, possibly up to 4 on some architectures.

The microoperations are gradually becoming more and more complicated to take advantage of the CISC design. Traditionally, RISC architectures had an advantage of being able to execute one instruction per cycle (in one execution unit), while CISC, with their complicated instruction set, had to use slow microcode.

Now, when memory access is growing into the major bottleneck, it becomes an advantage of the CISC design to have small code memory footprint. Also, as advances in processor design allow a more complicated execution unit logic, even some of the more complex instructions can be executed in one clock cycle. The RISC architectures often have to use more instructions to realize the same effect as a more sophisticated CISC instruction, which is a further advantage of CISC.

In the Pentium M design, for example, a memory-write operation uses only one (fused) microoperation. In previous designs, it has been split into two microoperations, one that calculates the address and another that writes the data.

2.1.3 Register renaming

The x86 architecture only has 8 integer and 8 floating point registers, which limits the amount of data that doesn't have to be loaded and stored to memory or cache. Since Pentium Pro, processors actually have many more physical[14] registers and use a technique called register renaming. They assign to every operation input and output registers from all the microarchitectural registers and store a mapping between these registers and the small number of virtual or architectural registers in a special *register alias table* (RAT).[15]

13. Microoperations are sometimes called μops or macro-operations. If they are constructed by merging other microoperations together, they are sometimes called fused μops.

14. Physical registers are also called microarchitectural registers, they represent a physical memory available in the microarchitecture but not directly accessible from code.

15. The register alias table is called Integer Future File and Register File on AMD processors.

This also reduces false dependency chains by reducing register reuse – the same architectural register may be mapped to different physical registers when there is no dependency.

2.1.4 Instruction scheduling and reservation station

Since Pentium Pro, microoperations are queued in a *reservation station* (RS) before being executed to allow out-of-order execution. A scheduler dispatches them from the RS to execution units through *ports*, when all input dependencies are ready – either calculated by previous operations (using fast store-load forwarding) or fetched from cache or memory. Different execution units or ports may have additional scheduling requirements, complicated operations, such as a multiplication or division can often not be executed in every execution unit or go through every port.

The scheduler in Pentium 4 is different in that it schedules operations speculatively, estimating the time when all input dependencies will be ready. When the estimation fails and dependencies are not yet ready, the execution is repeated again.[16]

2.1.5 Reorder buffer and retirement

Instruction retirement is the write-back of the results of the operation to its final destination and altering the user-visible processors state. This must be done with caution, in order to always present a defined state to the exterior.

On Intel microprocessors, before the microoperations go to Reservation Station, they go through *reorder buffer* (ROB), which records information for them.[17] After they are processed in the execution units, they return to ROB which ensures that all operations retire in order. The ROB also ensures that interrupts and exceptions behave in the same way as if they interrupted a sequential stream of in-order instructions. It also reads values of all operands for an instruction from physical registers. The size of the ROB, together with the size of the RS, determines how many instructions can be in flight at a time.[18]

The separation of physical and architectural registers using renaming and the concept of retirement allows speculative execution. As long as an instruction is not yet retired, it doesn't have an effect on anything else but the other instructions behind it, which are not retired, either. It is therefore possible to speculatively start executing code after a conditional branch instruction even if it is not clear yet if the branch will be taken or not.

2.1.6 Latency and throughput

Execution units have two important properties for a microoperation: latency and throughput. Latency of a microoperation is the number of clock cycles from the start of the operation until the operation is finished and the result is available. Throughput is the number of clock cycles it takes the microoperation from the start of the operation until another microoperation can enter the same execution unit.

16. One reason for failed estimations may be a cache miss. The cost of replaying an operation can be high.

17. This applies to Intel processors. AMD processors use a similar technique for out-of-order execution, but things are a little more complicated as three independent pipelines are used.

18. The number of instructions in flight is sometimes also called an instruction window.

In most modern processors, simple operations such as addition or bit operations have a throughput of one operation per cycle and a low latency, such as 1 to 3 cycles. Most execution units except division are usually fully pipelined, so they have a throughput of one operation per cycle. Only the most complicated operations, such as division, are often not pipelined and take up to tens of cycles.

Multiple microinstructions can usually flow from one stage to another every clock cycle. Some paths are wider than other and between some stages a buffer is used which allows for some flow delay compensation. If the pipeline is stalled for some time at one stage, accumulated microoperations from following stages can continue to flow and fill the gap created by the stall. This is of course only possible if there is some buffer space where the microoperations can accumulate.

2.2 Branch prediction

The pipeline length varies from processor to processor. When a branch instruction is encountered in the instruction fetch or decoding stage, it is not yet known if the branch will be taken or not and if taken, then to which target address.

Waiting for the target address to be ready would require, for direct jumps, the branch instruction to be completely decoded. To decide for conditinal jumps whether they will jump, and for indirect jumps also to determine the target address, it might be necessary to wait for retirement of all previous instructions.

The processor solves this problem by predicting if the jump will be taken and to which address. It then starts speculatively executing instructions following the branch instruction (in case it is predicted not to jump) or at the branch target (if it is predicted to jump).

In case the prediction was correct, no cycles were lost and the pipeline is fully used. If the prediction was not correct, it is necessary to throw away the results of all speculatively executed instructions and restart from the correct address. It usually requires the whole pipeline to be flushed, often waiting for all speculatively executed instructions to reach retirement (but not retire). The cost of the pipeline flush corresponds roughly to its total length as one stage usually takes one clock cycle.

Processor type	*Misprediction penalty*
Pentium MMX (U-pipe/V-pipe)	4/5
AMD K6	4
Athlon XP	10
Athlon 64	12
Pentium Pro, II, III	11 (10 – 12)
Pentium M	13
Pentium Core	14
Pentium Core 2	15
Pentium 4 – Northwood	20 (21)
Pentium 4E – Prescott, D, ...	23? (31)

Table 1. Table of branch minimal misprediction penalties.

Typical minimal misprediction penalties are shown in *Table 1.* Values in parenthesis are from different sources.[19]

For Pentium 4 and Pentium 4E, additional approximately 8 cycles need to be added for instructions that are not in the trace cache and have to be fetched from the L2 cache. This happens slightly more often than a L1 code cache miss on other processors, because of the lower trace cache efficiency. For Athlon processor, the penalty is 1 cycle more if the code segment base is not 0 (which is now rare).

The AMD K6 had a 6 stage pipeline and a branch misprediction was not that important (but was quite good, nevertheless). But as the pipeline gets longer, branch misprediction penalties are also higher. On Pentium Pro/II/III the misprediction penalty is usually about 10 to 20 clock cycles. On Pentium 4E processors, which have the longest pipeline, the minimal penalty is around 23, but typically about 45 cycles, and can be much higher, well over 100 clock cycles, if slow instructions, such as division, are executed. This is due to the fact that all microoperations need to get to retirement stage before pipeline can be flushed.

The need for a accurate branch predictor is therefore high. For branches that have not been taken recently and branch history is lost, a simple heuristics is used to determine if the branch is likely to be taken. The simplest algorithm is to predict branch as taken if and only if it points backwards, a more complicated solution also uses the distance to the target. This is called a static prediction and is usually only used when there is no other information about the branch available.

Processors usually have a branch target buffer, or BTB. It is a cache that stores information about branch behaviour and a branch target address. It is usually addressed using a part of the address of the control transfer instruction it predicts.

However, the lowest bits are sometimes not used for addressing, because instruction data is usually fetched in bigger chunks, and the instructions in the chunk are not yet decoded. The important thing is to decide which chunk to load next as soon as possible, before decoding it. For example, if instruction data is fetched in 16-byte chunks, the BTB may be addressed by bits 4..9 of the address of the last byte of a control transfer instruction. All branch instructions in a 16-byte aligned block, where all addresses differ only in the lowest bits 0..3, thus share the same BTB entry. This method is only used in some processors.

The simplest predictor is a local *saturating bit counter* for every branch, stored within a BTB. Each such counter is a linear automata with states ranging from *strongly not taken* to *strongly taken*, updated after the outcome of a branch.

A more sophisticated approach is a *two-level adaptive predictor*. It maintains a branch history of last n branches and uses it as an index into a pattern history table.[20] An entry in the table contains a simple predictor, such as the *saturating bit counter*. This kind of predictor can perfectly predict periodic patterns with a period of $n + 1$ or less and longer patterns if all n-bit sub-sequences in the period are different.

19. Pentium 4E misprediction penalty is reported differently in various sources, the often cited minimum 31 cycles penalty is probably the pipeline depth, not misprediction penalty.

20. This index can be combined with branch address or other value using a hash function to more evenly distribute the entries and prevent some aliasing.

To allow recording longer history patterns, a *two-level adaptive predictor* with a global *history pattern table* may be used. It profits from the fact that a behavior of a branch is often correlated with behavior of other branches. This type of predictor uses a global history of last n branches, but with a bigger value of n, because there is only one global history pattern table and it can be larger. A look-up in the BTB table is performed in parallel with a look-up in the history pattern table.[21]

There can also be an additional local *agree predictor* which is stored locally in every BTB entry and specifies if the final result of the prediction should be based on the outcome of the regular prediction by another method, such as a two-level predictor, or if it should be the opposite. This eliminates problems with branches that share the same entry in the pattern table but behave differently.

Most processor also have a return address stack that records return addresses for `CALL` instructions so that a `RET` instruction target can be accurately predicted.

For typical loops, a regular branch predictor is always wrong at the end of the loop. A special *loop counter* can be constructed which predicts the branch by the number of previous occurrences of the branch. When the branch is first encountered, it calculates the number of iterations and in subsequent cases, if the code is identified as having loop behaviour, it predicts the same number of iterations of the loop. A loop counter is usually combined with another base predictor which is used when loop character is not detected.

It is also important to identify the target address of a branch, not only if the branch will be taken. For direct branches, the target can be known soon, when the branch instruction is decoded. But for indirect branches, the target is known only after the operation that calculates it has been executed and possibly retired.

In a BTB entry, a target address is provided, which helps in the cases the target is always the same. In the Pentium M predictor, the branch history stores not only if a branch was taken, but also what was its target. The history pattern table contains a reference to a specific BTB entry that holds the predicted branch target address. Every time a new branch target is used, a new BTB entry is created for it. This way, even indirect jumps to different target addresses can be predicted if the address correlates with previous branch history.

2.3 Differences in pipeline between processors generations

The pipeline differs considerably between processor microarchitectures. There was a split in Intel product line after the Pentium III processor. Pentium 4 concentrated on increasing clock frequency and extending the pipeline and used a completely new design. On the other front, Pentium M, which was based on the Pentium III design, went in a different direction. It focused on improving performance at the same clock frequency and a reduced power consumption. It now seems that the power-hungry design of the Pentium 4 microarchitecture was a dead end, because of the growing concern over power consumption and heat dissipation. The newer Core and Core 2 families are based on the Pentium M microarchitecture.

21. On some processors (Pentium M), the BTB may be partly addressed from the pattern table, so an additional step might be required.

2.4 Intel Pentium and Pentium MMX processors

In the original Pentium and Pentium MMX design, there is no reservation station or reorder buffer, the decoding front-end is tightly coupled with execution units and permits no instruction reordering. Other techniques, such as register renaming, don't make sense with such design.

The Pentium processor can provide some execution parallelism, albeit in a fairly limited way. It has two parallel pipelines called U and V, which can sometimes operate simultaneously – an instruction can "*pair*" with another one. Some can pair in either pipe, some can pair only if they are in the U pipe, some only in the V pipe. There is a list of instructions which are pairable in one or two pipes and rules for pairing in [25].

The original Pentium uses a simple saturating counter for branch prediction with one peculiarity in that it moves from *strongly not taken* to *strongly taken* after one taken branch. Pentium MMX uses a two-level predictor, similar to mechanisms used by Pentium Pro, II and III.

When an address used in an instruction depends on the result of a previous calculation, the pipeline is stalled for one clock cycle to calculate the address. This is called an AGI (Address Generation Interlock) stall.

2.5 Intel Pentium Pro, II and III processors

The architecture of the P6 processors – Pentium Pro, Pentium II and Pentium III – is rather different from the previous Pentium MMX in that the front-end is detached from the execution core and the processor can therefore execute microoperations out-of-order.

The P6 family processors have a two-level adaptive predictor with a 16-way 512-entry BTB and a 16-entry return address stack to correctly predict 16 successive return addresses.

The Pentium Pro fetches instruction data in aligned 16-byte raw blocks into a buffer, one block each cycle. The buffer can hold two such blocks. From there the data are passed to an instruction decoder in instruction fetch blocks, or *IFETCH* blocks, which are up to 16 bytes long and start at an instruction boundary, except in some cases after taken branches. The next *IFETCH* block can be generated after the instruction lengths and the end of the last instruction in the previous block have been determined. The next block will start at the beginning of the last instruction not fully contained in the previous block.

The instruction fetch unit can be a bottleneck in fetching around (predicted) jumps. If both the *IFETCH* block containing the branch instruction and that containing the target of the branch span two aligned 16-byte raw blocks, the fetch unit is stalled for two clock cycles, because it needs to fetch two raw blocks to generate the *IFETCH* block of the target instruction for the decoder. If only one of them is not aligned, only one clock cycle is lost. It may therefore be advantageous to align targets of frequent jumps at a multiple of 16 if instruction fetching is a bottleneck.

Instruction length decoder unit reads data from *IFETCH* blocks and determines instruction boundaries. It is able to detect lengths of up to three instructions in one clock cycle, so instructions can be passed to three parallel decoders. It is also used to decide where to start the next instruction fetch block.

There are three instruction decoders D0, D1 and D2. Only D0 is able to decode more complicated instructions that are either longer than 8 bytes or generate more than one μop. The D0 decoder can generate up to 4 μops per clock cycle, which is enough for most instructions.

The first instruction from an *IFETCH* block always goes to D0. If one of the next two instructions would need more than one μop, the decoding is stalled until D0 is vacant. To prevent the stall, instructions should be preferably organized according to the 4-1-1 rule, which means that between instructions that generate more than 1 μop, there should be 2 simple instructions generating only one μop each.

The *IFETCH* boundaries can break the 4-1-1 rule if they occur after the first or second instruction in a 4-1-1 pattern, because the first instruction in the fetch block has to go to D0. It is difficult to predict where the boundaries will be, but in certain cases, it can be predicted. If a target of a jump is 16-byte aligned, the *IFETCH* block will always start at the beginning. After a misprediction, which always occurs after a loop, an *IFETCH* block will start at the nearest multiple of 16. Also, when two consecutive instructions have more than 16 bytes combined, the second instruction will always start a new block.

To allow out-of-order execution, a *reorder buffer* (ROB) with 40 entries is used. In this buffer, instructions are prepared for out-of-order execution and they have an entry stored there until they retire in correct order. The ROB also prepares values of input registers for all operations. If the operation that modified the register has been executed recently, the value is read directly from the not-yet-retired ROB entry of the operation. If is has been executed earlier than about 3 or 4 cycles, it has to be read from a permanent register file using one of the two available ports. A μop can have two input registers, so if none of them was modified recently, the throughput of the ROB stage is limited to only one μop per cycle.

The *register alias table* (RAT) is able to process up to 3 μops and rename up to 3 registers every clock cycle, but doesn't have other limitations and can even rename the same register three times in one cycle. The RAT has 40 physical registers which are mapped to architectural registers. The larger number of registers is needed to support speculative execution and to remove false dependencies.

The microprocessor generates 1 μop for simple operations on registers. For memory writes 2 μops are used. The first one is a store μop and the other one calculates the operand address. For *read/modify* instructions, also 2 μops are generated, one is a memory load operation and the other is the arithmetic operation. The more complicated *read/modify/write* instructions require 4 μops. The first calculates the memory address and loads the value, the second μop calculates the result, and writes registers and possibly a temporary value for the store, the third one reads the temporary value calculated in the preceding step. The fourth μop calculates the memory address again and stores the result. `PUSH` generates 3 μops, `POP` 2 μops, `CALL` and `RET` 4 μops each.

There is a stall called *partial register stall* which affects code that writes a small part of a register and then reads a bigger part of the same register. This stall occurs because smaller parts of a register are renamed to different temporary physical registers to prevent a false dependency between, for example, AH and AL. If the stall occurs, the μop that needs the data has to wait for the μop that wrote a part of the value to retire, which gives a delay of 5 to 6 cycles. This problem can be solved by using `MOVZX` instructions to read small values from memory or by `XOR`-ing the whole register with itself. The processor will mark that the register is 0 and when a small part is modified, it can be padded with 0s to make a bigger value without a delay.

A problem called *partial memory stall* is similar to a partial register stall and occurs when reading a bigger part of memory that includes a part that has been recently written to. It also occurs when reading a smaller part of a memory that has been written to if they don't start at the same address. The stall may also occur when writing and reading different addresses that happen to share the same set in the data cache. These stalls usually incur a delay of about 7 or 8 clock cycles.

Another stall called *partial flags stall* occurs with instructions using parts of the flags register after instructions that modify some of the flags but not all bits that are used. It occurs most likely with `LAHF`, `PUSHF` and `PUSHFD` instructions which read all flags, and gives a delay of about 4 clock cycles. A similar stall also occurs when reading flags after shifts and rotates by more than one.

2.6 Intel Pentium 4 processors

Pentium 4 has a very different design from previous Intel processors. The Prescott core supporting the EMT64T technology, sometimes abbreviated as Pentium 4E or P4E, introduced many changes in the design – most notably support for the 64-bit x86-64 instruction set introduced by AMD, even if it was disabled in early models.

Pentium 4 architecture introduced a new concept called *trace cache* to improve the instruction decoding stage and also branch prediction, both of which were, and still are, bottlenecks in many cases. It replaces the code cache that is used on other processors and is usually much larger but uses space less efficiently.

Instruction stream which is decoded is sent down the pipeline and at the same time stored in the trace cache. The sequence of instructions is stored there as a continuous stretch of microoperations, even across conditional branches in the original instruction stream. This allows to avoid the expensive instruction decoding and operate as a RISC processor, because the microinstructions stored in the trace cache are usually simple RISC-like instructions. It also helps with branch prediction as it feeds the pipeline with a continuous sequence of instructions.

Trace cache lines[22] are sequences of up to 6 decoded μops (microoperations) that can cross conditional branches. The lines are addressed by their virtual address[23] and possibly also by a set of last n branch decisions bits (taken/not taken). This

22. Multiple trace cache lines can be linked to make a *trace*.

23. Conversion to physical addresses doesn't need to be done until working with the L2 cache.

allows to store different trace paths starting at the same instruction but predicted to behave differently on branches. The same sequence of microoperations can be stored at different locations in the trace cache if there are multiple execution paths leading to the same code.

There is a separate trace cache branch predictor with a BTB in addition to a front-end branch predictor similar to the one used by previous processors. When traces of the code that is being executed have been built, the front-end branch predictor is not used and only the trace predictor is active.

Decoding on the Pentium 4 is quite slow, it can only process one instruction per clock cycle, but when a code has been executed recently, microoperations are not decoded from instruction stream but read directly from trace cache. Trace cache can deliver up to 6 μops every second clock cycle, which corresponds to the maximal throughput of the rest of the pipeline of about 3 μops per cycle.

More information about trace cache can be found in [32], [33] and [34].

The Pentium 4 has 4 execution ports that forward microoperations to execution units. Each execution unit is connected to only one port. Ports 0 and 1 have both an ALU[24] that operates on double of the base frequency of the processor.

The execution units operate on various speeds with some accepting two operations per clock cycle, some only once per two clock cycles and some, such as floating point division, have a much higher latency because they are not pipelined.

Unlike in previous architectures, register renaming is not used to differentiate between non-overlapping smaller parts of a register but the whole register is updated by every instruction that modifies a part of it. This creates false dependencies and sometimes requires an extra μop when accessing a part of a register but avoids stalls from synchronization of various parts.

There is a penalty of one clock cycle when mixing dependent instructions not executed by the same execution unit.

Some instructions which move floating point, MMX or XMM registers have a long latency of 6 cycles on Pentium 4 and 7 on Pentium 4E.

There is also a large penalty of about 10 to 20 cycles on Pentium 4 with a write to memory followed by a read from the same address if the write has not yet been executed because the value is not yet ready. The processor will speculatively try to read the value from memory over and over and cause a large delay.[25] The Pentium 4E processors are not affected by this problem.

The retirement unit can handle 3 μops per cycle but taken branches must retire in the first slot. Small loops where retirement can be a bottleneck should therefore preferably have a number of μops a multiple of 3.

24. Arithmetic-Logical Unit, performs integer operations, such as addition or bit operations.

25. The main reason for the delay is that other μops, which depend on the speculatively read values, will be repeatedly executed and discarded before retirement.

2.7 Intel Pentium M and Core processors

The microarchitecture of Pentium M processors is similar to the architecture of the P6 family (Pentium Pro/II/III). The Intel Core processors are very similar to the Pentium M processor but can use the 64-bit x64 instruction set,[26] like the Pentium 4E variants.

The pipeline is 3 or 4 stages longer and the processor has some new features, such as an improved branch prediction or so called *μop-fusion* that allows merging of two μops that were executed separately in previous processors.

There is a more complicated branch prediction mechanism on the Pentium M processor than on previous processors. It uses a two-level predictor combined with a loop counter. The BTB is 2-way associative and contains only 128 entries. It is quite small, so entries often push each other out, but the prediction mechanism works well on smaller hot-spots.

A loop counter is stored in every BTB entry and can predict repeated executions of loops with up to 64 iterations. There is a 2-bit saturating counter for detecting whether a branch has a loop behaviour or not. For branches that don't have a loop behaviour (the loop behaviour flag is not marked in its BTB entry), a two-level predictor with a global branch history table is used. Based on the outcome of the last 8 branches, an entry in the global history pattern table is selected to predict the jump.

The branch prediction is also able to predict indirect jumps to many different targets. More than one bit is recorded into the branch history table to differentiate not only if the branch was taken or not, but also what was the target of the branch.

There is a also a new *loop buffer* that can cache predecoded instructions for small loops that fit into 4 blocks of 16 aligned bytes.

Instruction fetching is improved, so fetching after a predicted jump is not delayed if the jump is not aligned – the *IFETCH* block will always start at the target address. This reduces the delays in fetching and makes *IFETCH* boundaries more predictable. Rules for decoder selection and throughput of the decoding unit are similar to those in the P6 processors.

Thanks to a method called μop-*fusion*, the throughput of the pipeline can be higher than 3 μops. The processor is able to join together some μops in the pipeline, which were separate in previous designs. The *fused* μops use only one entry in the ROB and in most other stages of the pipeline. They are separated again into the original μops just for the execution stage, where both of them can go to different execution units.

A memory write operation which was split into two μops in P6 processors is now joined back to only one fused μop. It is executed as two separate operations that always go to different units. Most *read/modify* instructions, except those using XMM registers, now also translate to one fused μop.

26. The x64 instruction set is sometimes called x86-64. It is a name for either the AMD64 or Intel's EMT64, which are very similar.

The ROB has been updated to allow a μop to have 3 input dependencies and to read 3 registers per clock cycle from the permanent register file. These modifications were necessary because a fused μop may now have up to 3 inputs.

To reduce dependency chains on the ESP register, a *stack engine* is used, which reduces the need to frequently update the ESP stack register. It stores a delta offset ESP_d needed to obtain the true ESP value in some of the common μops that manipulate stack, such as `PUSH`, `POP`, `CALL` and `RET`. These instructions therefore don't need to wait for previous updates to the stack pointer.

Other instructions that read the stack pointer but don't use the stack engine, such as `ADD ESP,4` or `MOV [ESP],EAX` will need a special μop inserted before them to update the value of ESP by ESP_d and set the delta offset ESP_d to 0. This μop is also added when the 8-bit ESP_d is near overflow. It has a latency of 1 clock cycle. A special table is used to undo the effects of the stack engine in case of a branch misprediction or an exception. To prevent the decoders from inserting the synchronization μop, instructions using the stack engine shouldn't be mixed with instructions not using it.

The `POP` and `PUSH` instructions translated on earlier processors to 3 and 4 μops, respectively, because they used the ALU for adjusting the stack pointer. Now, thanks to μop fusion, they use only one μop. The two store microoperations of `PUSH` are now fused together and the modification of the stack pointer is done by the stack engine without additional cost.

Memory store instructions may translate to 1 or 2 μops, depending on use of the SIB byte.[27] If the SIB byte is not used, that is if there is no scaled index register and the base register is not ESP, the instruction uses only 1 μop. Memory load operations don't have such issues and always translate to 1 μop.

The Pentium M processor has 5 execution ports that forward μops to execution units. Ports 0 and 1 are for arithmetic instructions, memory read μops go to port 2 and memory writes are split into two operations which go to ports 2 and 3, respectively. Some execution units, such as the floating point addition unit, are connected to both ports 0 and 1. Unfortunately, when using many instructions that go to an execution unit connected to two ports, one of the ports will be stalled most of the time, because the scheduler is not aware of the fact that two operations, which go to different ports, might need the same resources and schedules them to occupy both ports, even if they cannot execute in parallel and another instruction could go to one of the ports instead, without a stall.

The retirement stage can process 3 μops per cycle but taken branches can only retire in the first slot in the retirement station. If a branch instruction is not the first of the three μops that may come to retirement (instructions retire in order), it will have to wait for the next cycle to retire.

On some models, partial register access generates a similar stall as in the P6 architecture, but on newer models, a special μop that joins the different parts of a register is inserted where necessary, which reduces the delay from 5 or 6 to only 1 or 2 clock cycles.

27. A SIB (Scale-Index-Base) byte is added to the instruction encoding to support more complicated addressing modes, such as when a scaled index register is used.

2.8 Intel Core 2 processors

The Intel Core 2 processor is similar to the Pentium M design, but contains some improvements. It supports the x64 instruction set and has a higher maximum throughput of 4 μops along the whole pipeline instead of 3 in the Pentium M. The pipeline was upgraded and is 128-bit wide.

The instruction length decoder is also more advanced, it can decode up to 6 instructions in up to 16 bytes per clock cycle. Predecoding of next block cannot start until all instructions in the previous 16 byte block have been predecoded. If there are 7 instructions in the 16-byte block, it will take two clocks before the block will be predecoded and next block can be fetched.

There is a loop buffer between the predecoder and the decoder, similar as in the Pentium M, which allows to reuse predecoded instructions from previous runs. The loop buffer is only 64 byte long and can store four aligned 16-byte blocks.

There are 4 decoders, which can decode together up to 4 instructions per clock cycle, but only the D0 decoder can handle more complicated instructions. The decoders can provide up to 7 μops per clock cycle if the instructions use the 4-1-1-1 pattern. This means that there are 3 simple instructions generating only one μop each between instructions generating up to 4 μops. There can be an unlimited number of prefixes without penalty if the total length doesn't get over 15, but prefixes that change instruction length require more time to decode and the penalty can be up to 6 cycles.

Decoders D1–D3 can only generate a single μop each, but it can be a complex fused μop which is treated as a single μop for most of the pipeline except in the scheduler for the execution units. Fused μops can also be produced for instructions using XMM registers and for *read/modify/write* instructions which now only produce 2 μops instead of four, because the two *read/modify* μops are fused and the two write μops are also fused.

Even two μops that originated from different instructions can now be joined together, using so-called *macro-op fusion*. For example a `CMP` or `TEST` instruction can now be fused with the next conditional instruction to form a compare-and-branch macro-op. Such macro-ops are not split even for the execution stage and are treated as one μop during the whole pipeline. There are, however, several conditions that limit the use of macro-op fusion. For example, it doesn't work in 64-bit mode,[28] requires certain compare, test and jump instruction forms, there cannot be any instructions between the two operations which should be fused and there are also requirements on the alignment and position in a decoder block.

The ROB still has the limitation of only 3 read ports to the permanent register file. Reading the value from the permanent register file is necessary if the register hasn't been modified in the last 6 clock cycles. It is however difficult to predict such stalls because it is difficult to predict which μops will arrive together to the ROB.

28. The reason behind this is probably that 64 bit μops occupy more space in the ROB and there is no room for additional information required for fused macro-ops.

There are 6 execution ports, ports 0, 1 and 2 are for arithmetic instructions, port is 3 for memory read μops, ports 4 and 5 for memory write instructions. Most execution units can work with the full 128-bit length data. All arithmetic ports have their own ALU which can handle 128-bit moves and all except one also a 128-bit addition. There is one integer multiplier, one floating point multiplier, one jump unit and one shared division unit. The division unit is the only execution unit which is not pipelined.[29]

There is a delay of one clock cycle when using a result from the integer unit as input for the floating point unit or vice-versa.

A false dependency may be created when using different parts of the same XMM register, because no renaming to differentiate the parts, which is used for general purpose registers, is possible for XMM.

The data cache in Core 2 processor can simultaneously prefetch two data streams into the L1 and L2 cache. It is not possible to read and write memory with addresses which have the same bits 4 and 5 in one cycle – this is called a *cache bank conflict*. There are delays of around 12 cycles for reading across a 64-byte L1 cache line boundary and 10 cycles for writing across this boundary. Misaligned reads after a write to the same address have a 7 clock penalty. The access time for L1 and L2 caches are 3 and 9 clocks, respectively, which is quite fast.

2.9 AMD Athlon and Athlon 64 processors

Athlon and Athlon 64 processors[30] use a pipelined out-of-order design, as did the previous AMD K5 and K6 families, but the microarchitecture is different. Athlon 64 additionally supports the x64 instruction set, but is similar in design to the original Athlon.

Unlike Intel microarchitectures, Athlon uses three parallel pipelines from decode units until retirement. Some instructions are translated into macro-ops,[31] which may consist of two microoperations each, one for data loading and another for performing the actual operation. The maximum throughput of the pipeline is about 3 macro-ops per clock cycle.

Athlon and Athlon 64 processors use a two-level adaptive predictor with a global branch history of 8 jumps, a 4096-entry branch history pattern table, and a 2-way 2048-entry BTB.

The BTB is connected to the L1 code cache. For every aligned 16-byte block in the L1 cache, there are nine branch selectors for storing information about branches. These selectors contain information if the branch has never jumped, always jumped or if dynamic prediction or return stack should be used. All branches, except those never taken, also need a BTB entry.

29. This means that the throughput of microoperations is equal to the latency of the unit.

30. Athlon processors are also called the AMD K7 family, Athlon 64 are also called K8. Athlon/64 will be used to denote both Athlon and Athlon 64.

31. They are conceptually similar to fused macro-ops in Intel processors.

The BTB is organized as 1024 blocks with 2 entries each. Every aligned 16-byte block is normally assigned one BTB block based on its address, which means it can use 2 branch target entries. If a 16-byte block needs more entries, one additional entry can be borrowed from another block, possibly using a LRU algorithm to select the index. With every such allocation, however, an additional BTB entry is lost.[32] A 16-byte code block cannot have more than 3 BTB entries assigned. If there are more branches in one block that can all be taken, they steal each other a BTB entry and cause a misprediction every time.

Another problem is with the `RET` instruction, which is usually only 1 byte long. The branch selectors in the L1 code cache are organized in a way that the `RET` instruction may share a branch selector with another immediately preceding branch instruction or, in certain situations, the branch selector for `RET` won't be loaded at all. Both these problems will cause a misprediction.

Branch selectors are preserved when the code is evicted from the L1 code cache to the L2 cache and back, but the BTB entries are lost. If the branch target is later needed and there is no need for dynamic prediction, the target can be read from the instruction with a much lower penalty than if misprediction happened. Indirect jumps will always cause a misprediction because the branch target cannot be calculated from the instruction.

There is also a return address stack, which contains 8 entries for branches that have a flag in the branch selector indicating return address stack should be used.

The instruction fetch unit always fetches one 16-byte aligned chunk from L1 code cache per clock cycle in a 32-byte decode buffer. After a predicted jump, two cycles are needed to fill this buffer before decoding can start. The decoding can be a bottleneck, especially if the code contains many jumps, because every taken jump has a minimal overhead of 2 cycles.[33] Non-aligned jumps can have a bigger overhead because a big part of the first 16-byte chunk may be wasted if the target is located near the end.

Instruction length decoder can process only one instruction per cycle, but instruction boundaries are stored in the L2 cache, so the length decoder speed is not so critical. There is even more information in the L1 cache, which help in the decoding process, such as the type of instruction or location of the opcode in the instruction encoding.

There are 3 decoders and each of them can process one instruction with a maximum of 3 prefixes[34] per cycle.

Instructions can be categorized into two groups: single or double instructions, which generate one or two macro-op, respectively, and vector path instructions which generate more than two macro-ops. Other decoders are stalled when a vector path instruction is decoded. The original 32-bit Athlon treats all instructions which generate more than one macro-op as vector path instructions.

32. A hypothesis exists to explain this behavior: one BTB entry is used as a pointer to the BTB block containing the next two entries.

33. This means that the maximum throughput for taken branches is one per two clock cycles.

34. It is rare to have more than two prefixes in normal 32-bit or 64-bit code.

Many instructions, such as *read/modify*, *read/modify/write* or complex operations with more than two dependencies, which require two μops on Intel processors, only use one macro-op.

Athlon has a 72-entry instruction control unit which serves a similar purpose as the reorder buffer (ROB) in Intel processors. It schedules macro-ops into the separate pipelines. There are 88 physical registers, a 18-entry integer reservation station and a 36-entry floating point reservation station.

Each of the 3 integer pipelines has its own address-generation unit (AGU) and ALU. The three ALUs can execute all operations except multiplication, which can only be handled in the ALU0. The double-size multiplication operations (using RDX for the high part) additionally use ALU1. The 3 floating point pipelines share three specialized execution units: FADD, FMUL and FMISC. All three execution units can handle memory loads, memory stores go to FMISC.

Scheduling in the floating point pipelines has several problems. Macro-ops that can go to two or all three floating point execution units are often scheduled to a unit that is not vacant or could be used by another macro-op, which cannot go to another unit. Another problem is that a macro-op cannot be scheduled to a floating point execution unit if it would finish at the same clock cycle as another operation in the same unit, which will occupy the single result bus. The scheduler is not able to redirect the waiting operation to another unit or start a different macro-op with a different latency instead.

All operations with MMX and XMM registers are executed in the floating point units. The execution units are only 64-bit wide, so every 128-bit instruction is decomposed into at least two 64-bit macro-ops. Using 128-bit instructions therefore only improves the decoding time, not execution.

Reading a 8-bit or 16-bit memory operand to a register produces a one clock penalty, because the original value of the high bits is taken as a dependency.

There is a delay when XMM instructions for different data types, such as packed double versus packed integers, are used in a dependency chain. This doesn't apply to instructions that only read/write memory and don't perform any other task, such as `MOVAPD`.[35]

A false dependency is introduced between using different smaller parts of the same register or even between writes to the same part of a register. The processor assumes that a write to one part is dependent on the contents of the previous write, even if it used the same part. This dependency can be eliminated by clearing the full register first, for example by a `XOR` with itself. This false dependency is also not generated when using a 64-bit low or high part of a 128-bit XMM register, as these are stored as two different registers anyway.

False dependencies can also be introduced between different parts of flags. Some operations are recognized as independent, if the flags written and the flags tested are from different groups of flags.

35. The `MOVAPD` and `MOVDQA` instructions may therefore be replaced by the one byte shorter `MOVAPS` for reading and writing memory.

Cache memory is accessible by two ports, both can be independently used for reading or writing. Cache lines are 64 bytes long and organized into 8 banks of 8 bytes. It is not possible to read from the same bank in different cache lines in one cycle.

Writes to cache memory proceed in order and before subsequent reads from the same address. Reads proceed in order unless the data are not in the L1 cache. In this case, reads may go in any order, as they arrive from L2 cache or main memory. No memory read or write operation can proceed before addresses of all previous operations are calculated.

2.10 Major bottlenecks and possible optimizations

Some code will work well on a whole range of processor microarchitectures but some constructs only help certain processors but hurt the performance on others. It is important to know where are the bottlenecks of a particular processor or of modern processors in general.

2.10.1 Memory access performance

Data memory access can be one of the biggest bottlenecks in code. Every L1 data cache miss may delay the execution by about 10 cycles and much more if the data is out of L2 cache as well. Especially when data needed in a dependency chain have to be loaded from the main memory, the processor is basically stalled.

Memory is cached in cache lines, which hold aligned blocks from memory, usually 64 bytes long. Alignment of important memory structures by 64 may help to keep a bigger part of the structure together in one cache line and reduce the probability of eviction from cache.

Another optimization involves storing values on stack, which usually contains a lot of frequently used data on its top, and therefore will probably be present in the L1 cache. Data that are used together should be stored together to minimize cache footprint and reduce probability of eviction of a part of the data from cache.

The Pentium 4 processor has a small 8kB or 16kB (on P4E) write-through L1 cache, which can therefore be a bottleneck. More effort in using the cache efficiently by alignment and packing data together may be necessary.

On Pentium 4, there is also a large penalty of up to 20 cycles for reading memory after a write to the same address, which has not been completed yet. This can happen for example in integer to floating point conversion: `IMUL EAX,10`; `MOV [mem32],EAX`, followed by `FILD [mem32]`. It can be improved by producing a false dependency of the load operation on the value, that is waiting to be written, to force a serialization. In this case it means inserting, for example, `AND EAX,0` before the load operation[36] and replacing the load by `FILD [mem32 + EAX]`. The `AND` instruction will not break the dependency chain of EAX and `FILD` will be delayed until the value of EAX is calculated, long enough to prevent the stall.

36. A `XOR` operation cannot be used, because the processor would understand that it is clearing the register `EAX` and breaking the dependency chain in it, which is undesirable in this case.

Memory prefetching can be used, either using the `PREFETCH` instruction or by loading the data into memory or using something like `CMP ESP,[mem]`.[37]

On Athlon 64, 128-bit memory reads and writes[38] are treated as two 64-bit macro-ops, which can be executed only in the FMISC execution unit. If memory performance is a bottleneck, read/modify instructions may be used instead, because they can go to either FADD or FMUL units, and two such operations can be performed in parallel.

For copying memory, it is better to use 64-bit general purpose registers, if they are available. When using the 32-bit mode, 64-bit MMX or floating point registers, which can go to any floating point execution unit, may be used.

For Athlon/64, data that are often used together shouldn't be spaced 64 bytes apart, because it could cause cache bank conflicts when reading both values at the same time. On Core 2, there is a similar problem when simultaneously reading and writing to addresses that fall into the same 16-byte block modulo 64, i.e. that have the same bits 4 and 5.

2.10.2 Code cache performance

Code cache[39] misses can be an important performance bottleneck if the frequently executed code, *hot-spot*, doesn't fit into the cache. Hot-spots may be reorganized to better fit into the cache by separating out less frequently executed code to other addresses and compacting the hot code together.

Alignment of the code also helps to keep the cache footprint smaller and decreases the chance that some of the code will be evicted from cache or that several addresses will fall into the same cache set and force each other out of the cache, if there are more such addresses than the associativity of the cache.

Pentium 4, which doesn't have a code cache, has less problems with hot spots that are distributed in a larger area, but there are other problems.

2.10.3 Trace cache

For the Pentium 4/4E processors, trace cache has to be taken into account. To save space in the trace cache, instructions which generate fewer μops should be preferred and conditional jumps should be replaced by conditional moves unless it would create delays due to extra dependencies.

On Pentium 4, but not 4E, operands and relative memory addresses should be preferably kept in the range between -2^{15} and 2^{15} in order to fit into the 16 bits of storage in a trace cache entry, which are reserved for operands of a μop. If it does not fit, space needs to be borrowed from another entry, possibly requiring allocation of a new entry.

---

37. The `CMP` instruction is used because it only changes flags. The ESP register can be used on processors which don't have a stack engine, because it is usually not a part of a long dependency chain. Other registers without long dependencies can be used instead on processors with a stack engine.

38. The 128-bit memory operations can be done only with a XMM register.

39. Code cache is also called instruction cache.

Small loops with branches inside can also limit the performance of the trace cache to less than 3 μops per clock cycle, because they may break trace cache lines.

2.10.4 Reducing branch misprediction penalty

A long pipeline has some advantages, but a significant portion of the pipeline may get flushed by a branch misprediction and cause a stall proportional to a significant portion of the pipeline length. Reducing this stalls is an important optimization goal.

One way to reduce mispredictions is removing some of the branches. Beyond increasing chances of a misprediction, code with many branches can also fill the BTB buffer and decrease the efficiency of the prediction mechanism for other jumps.

To reduce the number of branches, small procedures may be inlined and some conditional jumps may be replaced by conditional moves (`CMOVcc` instructions) or conditional sets (`SETcc` instructions).

Sometimes, conditional branches may be replaced by arithmetic calculation. For example in the standard signed integer arithmetic, the min functions can be written as $\min(x, y) = y + ((x - y) \gg (n - 1)) \,\&\, (x - y)$, where n is the number of bits in which the machine stores variables x and y, because $(x - y) \gg (n - 1)$ is either 0 if $x \geqslant y$ or has all bits set if $x < y$. Care must be taken, that the resulting code can sometimes be slower than if using branches, especially when the branches would be well predicted in most cases.

On Athlon/64 processors, BTB entries are lost for every aligned 16-byte block of instructions that contains more than 3 branches, except those never taken. If a block contains more than three branches, they will contest for the BTB entries and there will be a misprediction every time. Aligned 16-byte blocks should therefore never contain more than 3 branches and unless there are enough BTB entries, they should even contain at most 2 branches. A large number of branch instructions in small space is most commonly found when coding a *switch* statement as a series of conditional branches. It can be replaced by an indirect jump in this case.

Another problem with Athlon/64 is that the `RET` instruction, which is usually only 1 byte long, might share a branch selector with another branch. This happens if the `RET` instruction is at an odd address and there is a branch immediately before it. Similar problem, caused by not loading the branch selector at all, can occur if the `RET` instruction is at an even address not divisible by 16 and is either jumped at or immediately follows a mispredicted branch.

Both problems, which can cause mispredictions, can be solved by making the `RET` instruction at least 2 bytes long, for example by prefixing it with a `REPZ` prefix, which doesn't change its semantics. This should only be done if the `RET` instruction immediately follows a branch instruction or if it is on an even address not divisible by 16 and is a target of some jump.

2.10.5 Instruction fetch and decoding

Instruction fetching is a likely bottleneck in the more powerful Core 2 and Athlon 64 processors, because the decoders do not evolve as fast as the rest of the pipeline.

If decoding is a bottleneck, the 4-1-1 rule should be used for P6 and Pentium M and the 4-1-1-1 pattern for Core 2. This means that instructions which produce more than one μop should be interleaved by 3 or 4 simple instructions translating to only one μop.

The Pentium Pro/II/III processors have delays when a branch instruction crosses IFETCH boundaries or if the target instruction is not aligned to a multiple of 16. To prevent the stall, targets of a jump should be aligned to a multiple of 16. Pentium M and Core 2 don't need such branch alignment. This optimization can actually make the performance worse on these processors, because it results in code size growth, which can impact code cache performance.

Athlon/64 processors always fetch blocks aligned by 16, so targets of frequent jumps should not be near the end of a 16-byte aligned block, because the subsequent block would also have to be fetched, delaying decoding by one cycle. Decoding of tiny loops that cross a 16-byte boundary will also need one clock cycle more every iteration because a new block will have to be fetched every iteration. Alignment of important jump targets by 16 can be useful if decoding is a problem. This can be done using a `NOP` or similar instructions or by making previous instructions artificially longer, for example by using redundant prefixes.

Complicated instructions involving microcode should be avoided if decoding is a bottleneck, because they decode slowly on Pentium 4 and other processors. The decoding is, however, a bottleneck on Pentium 4 processors only if the critical parts of code don't fit into the trace cache. Otherwise, decoded μops are fed directly from the trace cache.

Using more than one prefix should be avoided, especially on the Pentium M processor. On Core 2 processors, additional prefixes cause no delays.

Small loops shorter than 64 bytes in four 16-byte aligned parts can be predecoded in the loop buffer in Pentium M, Core and Core 2 processors. Longer loops may be split into several 16-byte aligned loops shorter than 64, if decoding is the bottleneck.

For Core 2 processors, the instruction predecoder cannot fetch another 16 byte block until it has fully decoded the previous one. In case predecoding is a bottleneck, it can be better to make some instructions a little longer to fill the 16 bytes with just 6 instructions if there are 7 or 8 instructions in the block.

Core 2 and other processors have a large penalty when decoding operand size prefix (66H) or address size prefix (67H). It is therefore advisable to replace instructions such as `MOV AX,1` by `MOV EAX,1` in 32-bit and 64-bit code.

For memory operands, it is better to use `MOVZX EAX, BYTE PTR [mem8]` instead of `MOV AL, BYTE PTR [mem8]`, and likewise with 16-bit operands.

For the same reasons, using a 16-bit immediate operand in 32-bit code should be avoided because it uses the operand size changing prefix.

A code using 16-bit parts of a register and an immediate operand, which cannot be represented as a 8-bit signed integer, should be preferably replaced by a variant that uses the full register. For example `ADD AX,200` could be written using the full register as `ADD EAX,200`. Note that no change is needed for `ADD AX,100`, because the number 100 can be represented as a 8-bit signed integer and no operand size changing prefix is needed.

Some instructions that work with XMM registers come in several variants with different encodings, but doing the same thing. It is therefore preferable, for example, to use the shorter `MOVAPS` instruction rather than the longer `MOVAPD` or `MOVDQA` instructions. Care must be taken that some instructions, which could be replaced by a differently typed form,[40] might cause a delay on the Athlon 64 processor if they are mixed in dependency chains with other instructions typed for different data types[41] in XMM registers.

2.10.6 Breaking dependency chains

Long dependency chains often make it harder for the processor to perform out-of-order and parallel execution. This can cause delays in the pipeline, especially when dependency chains contain slow operations such as division, multiplication or some floating point operations.

Some processors, which don't rename smaller parts of a full register, can suffer from false dependencies. This can happen even between two writes to the same part of a register, because the processor treats such writes as dependent on the previous contents of the register. On Pentium 4, Core 2 and Athlon/64 processors, it is possible to use `XOR` or `SUB` of a register with itself as a way to break the dependency chain for a register.

On most processors, it is possible to use the `MOVZX` instruction instead of moving data to a part of a register. This also breaks a dependency chain, but this method is slower on Pentium 1/MMX.

2.10.7 Partial register, memory and flags stalls

Partial register and memory stalls occur when a write of a value is followed by a read of some larger part of register or memory containing the value that has just been written. The problem with writing to a part of a register and then reading a different dependent part, is that the parts have to be combined together, often requiring that the write is retired first. Partial memory access prevents effective store-to-load forwarding. The read operation therefore needs to wait until all parts of the value are combined together.[42]

40. For example, some bit instructions, such as `PXOR` with a XMM register operand, `XORPS` and `XORPD`, use a different encoding but perform the same operation.

41. The only difference that matters here is between integer operands and floating point operands. There is no delay when mixing instructions typed for packed single and packed double values, according to [25].

42. Similar problems may be experienced even for reads and writes with different addresses which have the same set value in the cache.

Partial flags stall is a similar problem which can occur when only some flags are written and other flags are then used, generating either false dependencies or requiring combining of the different parts of flags together.

Partial memory stalls are similar on most processors, partial flags stalls are less frequent on Athlon processors, because they keep flags in several different groups, which reduces false dependencies.

Testing flags immediately after shifts and rotates by more than one can cause a partial flags stall on most Intel processors. To reduce the stall, other instructions can be scheduled between the shift/rotate and using the flags.

Some processors (such as Athlon/64 and Pentium 4) don't have partial register stalls because they always keep the whole registers together, but this introduces another problems – false dependencies. The Core, Core 2 and some versions of Pentium M prevent partial register stalls by introducing special μops that combine parts of the register together, reducing the penalty to only 1 or 2 cycles.

On most processors, it is possible to work with full registers instead, for example use the `MOVZX` or `MOVSX` instructions instead of loading smaller data from memory only to a part of a register. These instructions break dependency chains and remove partial register stalls.

Another solution is to neutralize the full register before writing to a part of it, for example with a `XOR` operation of the register with itself.

Using the 8–15 bit parts of registers, such as AH, BH, CH or DH, gives a penalty on several processors including Pentium 4 and Core 2 architectures. On Athlon and Athlon 64, there is a penalty if AH, CH, BH or CH is written to memory that is soon afterwards read from.

2.10.8 Instruction scheduling and ROB bottlenecks

The Pentium Pro/II/III processors have only two ports in the ROB stage to read from the register file (physical registers). This can limit the throughput significantly. The Pentium M and Core 2 processors both have a limitation of 3 reads from a register file, which is also a likely bottleneck, especially on the Core 2 which can feed the ROB stage with more instructions each clock cycle.

This stall can be improved by scheduling μops so that most of the input registers are modified by preceding μops and can therefore be read directly from other ROB entries without accessing the permanent register file. Care must be taken not to introduce any slow instructions in the dependency chain, though.

The registers which are likely to cause this problem problem are the stack and frame pointers, loop invariants stored in a register or *'this'* pointer used in object-oriented languages to point to the current object, because they are all frequently read and less frequently written to. Scheduling instructions so that reads of a register are not too far from writes to the register can help. Another solution would be to store some values in memory instead, but it is necessary to take into account that there can be other bottlenecks introduced instead.

On Athlon/64 processors, macro-ops which go to a floating point execution unit must wait for the result bus to be vacant at the time the result will be ready. So if a longer latency instruction, which is not in the dependency chain, is executed first, a macro-op with a shorter latency cannot go to the same execution unit pipeline and can be postponed for a relatively long time. This bottleneck can be prevented if instructions generating macro-ops with a shorter latency are put before instructions generating macro-ops with a longer latency when they both go to the same execution unit.

To prevent delays caused by suboptimal scheduling in the floating point units, it is recommended to mix integer and floating point instructions for Athlon/64.

No memory read or write operation can proceed before addresses of all previous operations are calculated. It is therefore recommended to calculate values that are used in addressing as early as possible.

2.10.9 Using execution ports and execution units evenly

On Pentium M, it is sometimes advantageous to replace `MOV`s between registers and from immediate data to registers by moving data from memory, because it allows to shift load away from execution ports 0 and 1 to the memory load port 2.

On Pentium 4, some execution units operate on higher frequencies and have lower latencies than other units. If some μops go to a slower unit through a port that can feed μops to a faster unit, the port is limited by the latency of the slower unit, delaying other μops that could go to the faster execution units. It can help change the instructions that go to slower units using fast ports to other similar instructions that use a different port. This can be done for example with the `MOV` instruction which goes to a faster port when used with an immediate operand, but uses port 2 if loading a memory operand.

2.10.10 Optimizing execution units usage

On some architectures, such as Pentium 4 or Core 2, there is a penalty when mixing the use of integer and floating point units in a dependency chain, because of the need to forward data between different units. These data transfers between the units should be minimized – it is better to place integer and floating point instructions separately and then join the results as seldom as possible.

On Athlon/64, a there is a similar penalty for mixing instructions working with XMM registers that operate on differently-typed data, except instructions that only move data or read data from memory.

Code that uses a lot of multiplications on Pentium 4 might consider using MMX or XMM registers because the integer unit needs to transfer values for multiplication to the floating point unit and back.

2.10.11 Instruction selection

The Pentium 4 processors don't have a dedicated barrel-shift unit for fast shifting. Simple shifts, perhaps by up to 3 or 4, can be replaced by repeated addition of the same register with itself. The Pentium 4E doesn't have this limitation and shifts are as fast as on other processors.

Also the LEA instruction is slower on P4, because it is split into additions and shifts. Patterns like `LEA EAX,[EAX + EAX*4]/SHL EAX,1` for multiplying the contents of EAX by 10 are now considerably slower. Partly because of the `SHL` instruction, partly because of the inefficient `LEA` instruction.

On Pentium 4, the `INC` and `DEC` instructions are slower than using `ADD`/`SUB` and can also create false dependencies on the carry flag and cause partial flags stall.

Also on Pentium 4, some forms of memory store instructions which use the SIB byte produce more μops than others. If possible, the SIB byte should not be used frequently for memory store instructions. This means that ESP should not be used as a base pointer and there should not be a scaled index register for memory targets.

2.10.12 Taking advantage of μop and macro-op fusion

When using 32-bit mode, a combination of a `CMP`/`TEST` instruction with a conditional branch can take advantage of macro-op fusion to become a single fused macro-op. There are however other requirements, certain combinations of compare/test instructions are not possible, such as a test between a memory and an immediate operand. Additional requirements specify that the conditional jumps cannot test the overflow flag,[43] there can be no other instructions between the compare/test and the branch instructions, and there are other alignment requirements. If it is possible to meet all these requirements, code with a lot of branches can be significantly speeded up.

On Pentium M, it is sometimes better to use floating point registers instead of XMM registers if μop fusion can improve performance.

2.10.13 Reducing stack synchronization μops

On Pentium M and later Intel processors, it is sometimes faster to replace `MOV` instructions relative to the stack pointer by `PUSH` or `POP`. It can save some instruction length and can reduce the number of ESP_d synchronization μops.

Instructions such as `ADD ESP,4` to clean up after a function call can be replaced by a `POP ECX` (or similar) if the next instruction is a `PUSH`, `POP`, `RET` or another `CALL`, because no synchronization μops need to be inserted when all instructions use the stack engine.

2.10.14 Retirement limitations

On Pentium M and Core processors, three instructions can retire in one clock cycle, but taken jumps can only retire in the first retirement unit slot. Small, time-critical loops should therefore preferably have a length divisible by 3, if retirement is a bottleneck. The retirement units in the Core 2 and Athlon/64 architectures don't have such limitations.

43. The overflow flag is more difficult to calculate.

3 ELF file format and BFD library

The *IA-32 Binary Optimizer* framework works with binary files in the ELF format. I will briefly introduce binary file formats in general, the Executable and Linkable Format ELF and the BFD library, which is used to access them.

3.1 BFD library

The BFD (Binary File Descriptor) library allows manipulating many different object file formats in a unified way. It can even read, create and modify Windows PE files, so extending the optimizer to work with `.exe` files for the IA-32 Windows platform should not be too difficult.

The library works with handles to binary files and provides abstractions for many common concepts, such as sections, symbols, relocations or debug information. It provides a unified interface for most of the work in modifying binary program files. After the program is processed by the BFD library, some small fixes to the file are performed, but the ELF-specific part is very small, most of the work is done by BFD.

More information about BFD can be found in the *BFD Manual* [36].

3.2 ELF files

The ELF file format can be used for executable programs, dynamically linked libraries of for relocatable object files. It is used in many UNIX-like systems on several different 32-bit and 64-bit architectures.

The optimizer is currently only able to process binary executable programs. Support for dynamically linked libraries should not be too difficult to add. It might be even easier to analyse them, because they contain additional information which makes it possible to place the code anywhere in the address space. This information could be used to disambiguate code memory references and trampolines would not be necessary.

Processing relocatable files might also be interesting, especially for results of partial linking, because they also contain relocation information which could be used to disambiguate code addresses and improve optimization process, while not growing the resulting file size. The optimized relocatable file would then be processed by a linker to create the resulting executable file.

The focus of the ELF file format is on fast processing. All records in the file have a fixed length and structure.[44] Variable-length information must be stored elsewhere and referenced. Strings are stored in special string tables.

ELF files can be analysed with tools, such as `readelf` or `objdump`.

44. The structure is differs between 32-bit and 64-bit variants of the ELF format.

3.2.1 File header

Every ELF file starts with a file header which contains various information about the object file, such as a description of the architecture, and also location of other data, stored as offsets into the file.

The ELF files provide two different views on their content. One is represented by the concept of *sections*, which provides more structured and detailed information about different parts of the file. The other are *program segments* which represent a more run-time perspective of the file, focusing more on the properties of the image of the program in memory and fast loading.

There is a *section header table* describing sections and a *program header table* describing program segments. One of the entries in the program header table refers to the location of the dynamic section, which provides necessary information for a run-time loader and dynamic linker. Relocatable files most often lack program headers.

3.2.2 Program segments

Program header table is an array of simple structures that specify run-time behavior of the program. There are different types of program headers. The `DYNAMIC` program header points to the dynamic section (described later), which is used in dynamic linking. So called interpreter, usually a dynamic linker, is specified in the `INTERP` program header. The interpreter is called first, when the application is about to start. It performs some necessary actions, such as loading required libraries and relocating some of the content.

The last important type is the `LOAD` program header,[45] which describes program segments. These segments define how should data from the file be loaded in the process image in memory.

Each `LOAD` header specifies an offset and length of data in file, a virtual address, where the content should be positioned, and length in memory. If the length in memory is larger than the length in file, the rest of the pages not loaded from file are filled with zeros.

Segments should not overlap in memory, but their images often overlap in file. This is caused by the way in which the mapping of the pages from file is done. Memory management in most operating systems only allows to load pages from file which start at offsets aligned to page size. Code and data are loaded by two different `LOAD` segments and don't share pages,[46] but code and data sections are usually close to each other in the file, so two pages at different virtual memory addresses may load the same data.[47]

45. A `LOAD` header specifies what is also called a `LOAD` segment. It has nothing to do with processor segmentation, however.

46. One reason for this is page protection. It is possible on some processors to disallow execution of data pages or modification of code pages.

47. As a result, there is often a part of the beginning of data in the last page of the code segment and a part of the end of code in the first page of the data segment.

3.2.3 Sections

A binary object file is divided into *sections*, which are described in the entries of the section header table linked from the file header.

Every section has its name, various flags, offset and size in the file and an assigned virtual memory address.[48]

Most of the contents of an ELF file, except for the file header, section headers or program headers, is covered by a section. Sections may overlap by definition, but almost never do, because standard compilers or linkers don't produce such files.

Sections are used extensively by the compiler and linker. In relocatable object files, there is often a large number of sections used for special purposes.[49] Some of them contain executable code, some data and other additional information used for linking, executing or debugging.

Section name	*Description*
`.text`	Binary code of the program
`.data`	Global initialized data
`.rodata`	Global read-only initialized data
`.bss`	Uninitialized data, not loaded from file
`.init`, `.fini`	Process initialization and termination code
`.got`, `.got.plt`	Global Offset Table – used for dynamic linking
`.plt`	Procedure Linkage Table – used with `.got`
`.symtab`	Symbol table
`.strtab`	String table for `.symtab`
`.rel.plt`	Relocations in the `.plt` section
`.dynamic`	Dynamic section, used at run-time
`.dynsym`	Symbol table for dynamic loading
`.dynstr`	String table for `.dynsym`
`.eh_frame`	Exception handler frame information
`.debug_aranges`	Map from address ranges to debug information
`.debug_info`	Global debug information about for one compilation unit
`.debug_abbrev`	Abbreviations used in `.debug_info`
`.debug_line`	Line information used by debuggers

Table 2. Sections of a typical executable file

Typical sections in an executable program are presented in the *Table 2* above. The section names and their function is not strictly specified by the ELF definition, but most compilers follow this convention.

48. For relocatable object files, the virtual memory address is set to 0, because it is not yet determined.

49. For example, in C++, some functions can occur in multiple source files but should not be duplicated in the resulting binary. They are placed in individual sections and referenced in a COMDAT section. The linker can then merge all such sections and remove all of them except one, so there is only a single copy of the code. This kind of linkage is sometimes called "vague linkage".

The code of the program can reside in multiple sections, but for executable files, it is usually all concentrated in the `.text` section. This section is also the only one that is instrumented and optimized by the optimizer framework.

Initialization code, which should be executed before the main entry point, is placed in the `.init` section. It is used for example for global constructors in C++. Termination code, used for global destructors and cleanup code, is put into the `.fini` section.

Data of the program are stored in `.data` and `.rodata` and `.bss` sections. Symbol table is stored in `.symtab` with symbol names in `.strtab` and dynamic symbols in `.dynsym` with names in `.dynstr`.

For dynamic linking, there is a procedure linkage table (PLT) stored in the `.plt` section and a global offset table (GOT) in the `.got` section. For every function which is used in the program but defined in a dynamically linked library, a small part of stub code in the PLT is generated and a place to store the final address of the function is allocated in the GOT. This stub code in the PLT is statically linked to all invocations of the function in the code. Every usage of the function will result in calling the stub in the PLT.

When a stub in the PLT is invoked, it uses the address of the function in its GOT entry to indirectly jump to the function. When the stub is invoked for the first time, the entry in the GOT table will usually[50] direct execution to a resolver of the dynamic linker which will determine the correct location of the function and update its GOT entry. Next time the stub for the function is called, the control will be transfered to the function directly, using a single indirect jump.

The reason for a PLT is that a compiler, when generating code for a function call, cannot find out if the definition of the function will be in a dynamic library or in another object file. It therefore always uses a direct call. This is the reason why the GOT cannot be used directly, but a stub in the PLT has to be generated.

On the IA-32 architecture, on the other hand, it would be possible to use only the PLT with final addresses of functions directly embedded and relocated inside the stub, but this would prevent the often larger PLT to be read-only and shared between all process instances of the binary. On other architectures, there could be problems that absolute addresses cannot be directly embedded in code, or that they would require special alignment.

The debug sections `.debug_*` are only used for debugging, so they are normally stripped out for installation and distribution. Data stored there are usually in the DWARF2 debugging format. The *IA-32 Binary Optimizer* is able to transfer this data, especially the line information, into the instrumented or optimized binaries.

Tables used for exception handling are put into the `.eh_frame` section. The format of this data is similar to the DWARF2 frame data, which are often stored in the `.debug_frame` section.

50. In special cases, an early binding may be requested which fills the final address into the GOT for all entries before the program is started. This behaviour may be triggered by defining the `LD_BIND_NOW` environment variable on most systems.

The duplication of the information has two reasons: one is that the `.eh_frame` data can be smaller, because it only captures information on stack frames which may use exceptions, the other is that debug information may be stripped altogether from a final installation of the binary.

An executable program could work without any sections,[51] all that matters for the program loader and dynamic linker is in the program segments and in the dynamic section. However, compilers and linkers always keep sections for content that is left in the file, even when the binary is stripped.

The binary optimizer relies on sections to find out the area where the executable code resides. From the `LOAD` segments, the exact boundaries of code cannot be reliably determined. If necessary, a heuristic approach and control flow graph search could provide this information, but is was not considered important as all files I have encountered had sections.

3.2.4 Dynamic section

The most important data needed when executing or dynamically linking an object file are stored in a *dynamic section*.[52] It is usually stored in the `.dynamic` program section[53] and is also referenced from the program header table by a `DYNAMIC` program header.

The dynamic section contains information on required libraries and their versions, addresses of the init and fini code, usually stored in `.init` and `.fini` sections, a dynamic symbol table with its string table and dynamic relocations.

The dynamic symbol table and dynamic relocations are only used by the dynamic linker, so they only include those symbols and relocations, which are involved in run-time linking.

3.2.5 Symbols

Symbols are used to assign a name to an address or range of addresses in the process image of the program. In relocatable object files, where addresses are not yet known, offsets in the file are used instead of VM addresses. There can be symbols representing functions, variables or sections.

Some symbols may be defined and have an address associated with them and some may be undefined. Defined symbols reside in the file, undefined symbols are matched against a defined symbol in another object file during static linking or in a dynamic library during dynamic linking. The usages of an undefined symbol are connected, in a process called relocation, to the symbol that was found for it in another file.

Symbols used for dynamic linking are called dynamic symbols. For every entry in the GOT, a dynamic symbol with the name of the entry's function is generated.

51. There is a tool called sstrip, which is able to remove section headers from program file.

52. The dynamic section is, however, not an ELF "section" in the sense as defined before.

53. This is only a common practice, there is no need for any sections at all in an executable file.

3.2.6 Relocations

Relocations, or relocation entries, are used to *relocate* object files during linking and dynamic linking. It is a process of connecting usages of a symbol with its definition. Every relocation entry has one of the predefined types and references a location in the process address space[54] and a symbol.

During linking,[55] every relocation entry is processed and the location it represents is updated to refer, in some way, to its symbol.

The means of updating the location differ according to the type of the relocation. The value written can be a relative offset between some reference point[56] and the address of the symbol. Other possibilities include writing the absolute address of the symbol or the address of the GOT entry for the symbol.

In relocatable object files, there are often relocations which update locations in code, in data and in the debug information. In executable files, there are usually only relocations for the GOT table and possibly for some data,[57] because code is usually mapped read-only and is shared between all processes executed from the same file.

The dynamic linker processes relocations and fills the addresses of the imported functions in the GOT (either on demand or all at once) and addresses of imported variables in the data segments.

---

54. In relocatable object files, file offsets are used instead of memory addresses – for the same reason as with symbols.

55. The method is similar for static and dynamic linking. Relocations, which are used in dynamic linking, are sometimes called dynamic relocations.

56. Such reference point can be the location referenced by the relocation or, for example, the start of a section.

57. A dynamic relocation is often added for variables such as `errno`.

4 Program overview

In this chapter, I will introduce the optimization system *IA-32 Binary Optimizer*, which consists a part of this thesis. It is written in C++ language, uses *BFD* and *dietlibc* libraries and requires autoconf/automake for building. It is published under the GNU GPL license and the latest sources can be obtained from the following URL:

`http://sweb.cz/Alexandr.Kara/IA32_binopt/`

Some examples of using the framework can be found in *Chapter C*.

The C++ language was selected because it is fairly portable, has a good support for higher level constructs and abstractions, while maintaining efficiency and good control over the generated code. It also allows to easily use existing C libraries, like BFD or dietlibc.

The BFD library is used for reading and writing executable program files, because it allows higher level view of the file and manages the internal details on its own. Another advantage of using this library is that it supports many different file formats, including a.out, used in older UNIX and similar systems, or PE/COFF, used on Windows machines. This, combined with the fact that the optimizer doesn't depend on any other library except BFD, dietlibc and the standard C++ library, means that porting to other systems – either as a host or as a target – should not be too difficult.

The dietlibc library is used to support embedding of a small independent and self-contained initialization code written in C into the instrumented binary program.

More information related to the source code structure and description of important classes can be found in *Appendix A*. There is a Doxygen-generated documentation, which can be built from the source tree by running the `doxygen` program in the top directory of the project.

The system is split into several programs, which use the same infrastructure. There is an instrumenting program called `ia32bopt_prepare`, which prepares a program for optimization by inserting counters on all basic blocks and branches. The output of the instrumentation phase is a program which writes the counters to a counting file `${IA32BINOPT_BASE}/path/to/executable`.[58]

Another program of the toolchain, called `ia32bopt_optimize`, takes the instrumented file, reads the counters from the counting file, performs some optimizations and writes the result to the optimized program file.

Other programs, such as `ia32bopt_analyse` and `is32bopt_disassemble` can be used to analyse either instrumented or input files. The last program `ia32bopt_cpuinfo` shows some more detailed information about the processor it detects. Some default optimization options are selected according to the current processor, so it may be useful to see the details of the CPU interpreted by the framework.

58. If the IA32BINOPT_BASE environment variable is not available, `/tmp/counters` is used instead.

4.1 Reading input file and decoding

The first phase of both instrumentation and optimization is decoding of the input binary file. It is done in several steps described in this section.

4.1.1 Opening file

The BFD library is set-up using `bfd_init()`, then the input file, either the original or instrumented program, is opened using `bfd_openr()`, its symbols, dynamic symbols and relocations are read.

A test for programs using C++ exceptions is carried out. Code that uses exceptions may behave incorrectly[59] when executed from a different location, because the exception handler tables are not created for the new code section.

The check for exceptions currently works by testing for presence of an imported dynamic symbol `__cxa_throw`, which is a good indication of code using exceptions. It doesn't detect code that only catches exceptions from library functions, however. Updating of exception tables is scheduled to be added to the framework, so it should not be that much of an issue in future.

4.1.2 Analysing potential jump targets

All sections of the input program are processed and potential addresses into the code are gathered (in `ConversionHelper`::*analyseProgram()*). This is necessary to identify places in the program, that could be a target of an unexpected jump. Direct jumps that have the target address in the instruction operand are easy to predict.[60] The problem is with indirect jumps. The target address is usually stored either in the constant data or as a part of instruction encoding, potentially anywhere in the program.

The address of a jump may actually be computed, but this is very unusual and difficult to solve, so it is ignored. The only exception from this is the call inside glibc from *call_gmon_start()* to *gmon_start()*, which is explicitly handled in the convertor.

The detection of potential jump targets is done in two steps. All code sections are first scanned to find the range of code addresses and to identify the start addresses of instructions (in `ConversionHelper`::*analyseSection()*). The detected instruction starts are written into a bitmap.

Then, data sections are scanned for aligned values that fall into the code range and point to a start of an instruction. Similarly, code sections are scanned for similar values in immediate data or in displacement of the `LEA` instruction.

59. It will usually crash when throwing an exception.

60. Except for far jumps, but they are rare in 32-bit mode and almost never jump into the same code section.

The detected jump targets are stored into a bitmap (called `possibleTargets`) and will be used later, to add trampolines back to the optimized code. Addresses of symbols are also stored in a bitmap (called `symbolAddresses`).

Along with the bitmap, the location of the reference, which caused the address to be marked as a potential target, is stored with every target. It may be later used to avoid a trampoline.

4.1.3 Cloning code section

When code should be converted, the original version is left in its place and cloned to a new section[61] code is modified. Parsing of the basic blocks and all other steps are performed on the new (cloned) section. The cloning of the code section is done in `ConversionHelper`::*cloneCodeSection()*.

If only analysing the file, this step is skipped and data are parsed from the original section.

4.1.4 Parsing basic blocks

Basic blocks are parsed in `ProgramSection`::*parseBlocks()*. A bitmap of addresses where blocks should be forced to end is passed to the function. In the beginning, a single basic block is formed and instruction parsing is initiated. As parsing continues, blocks are split after branch instructions and calls, at targets of a branches or calls and at addresses specified as forced block ends.

For far jumps and jumps outside of the code section, a `Relocation` object is created, and the instruction is relocated before new code is written to a file.

If removing of empty blocks is requested (using `--keep-empty-blocks=n` option), blocks that end with `NOP` instructions[62] are trimmed or removed completely if nothing is left.

4.2 Writing resulting program

When all instrumentation or optimization work is finished, the code is prepared for writing. This is done in several steps.

4.2.1 Placing blocks

All blocks are assigned an address so that they do not overlap. If a command-line option `--condense-blocks` was specified, blocks are placed so that a next block starts immediately after the block before. This may improve code locality, but also disrupt alignment. Block placement is done in `ProgramSection`::*condenseBlocks()* and `ProgramSection`::*fixBlockOverlaps()*.

61. The original section is appended a suffix '.orig'.

62. Instructions which are used as multi-byte NOPs, such as `MOV EAX,EAX` or `LEA EDI,[EDI + 0]`, are also included.

The next step is done in `ProgramConvertor`::*finishChangingAddresses()*. All sections are placed in memory so they don't overlap, while honouring alignment requirements. Symbol and relocation addresses are updated. Unlike local symbols and relocations, which are moved with their blocks while code is changed, external symbols (represented by `SymbolInfo` and `RelocationInfo` objects) retain the original address and are translated only after all modifications to the code are finished. Local relocations (`Relocation` objects) are also performed (relocated) at that point.

If additional code from another object file needs to be added to the new program,[63] it is added into a new code section in *fillHelperSection()* in `prepare_counting.cpp`. The new code is relocated just after being copied to the new section.

If section end alignment was requested (using the `--page-align-section-end` command-line option), it will be aligned, possibly by filling the end it with `NOP` instructions. This is done in `ConversionHelper`::*fixupClonedSection()*.

4.2.2 Trampolines

Trampolines are jumps from original code addresses to corresponding locations in the new code. When all addresses are finalized, trampolines may be processed. Addresses where a trampoline should be added are marked in the `possibleTargets` bitmap.[64] If the `--trampolines-on-counters` command-line option is specified, all locations with a profiling counter will also get a trampoline.

Some addresses, where trampolines should not be placed, may be specified with the `--avoid-trampoline=`<*address*> option. Values in the data that reference such addresses are updated to point to the new location. There is a risk that the value will not represent a code address and will get updated by mistake. If such value is changed, it may result in program malfunction. That is why no addresses where to avoid trampolines are specified by default.

Trampolines are also not inserted on special symbols, such as *call_gmon_start()*, which uses a relative addressing of data outside of code and needs to be executed in the original code.

At addresses, where a trampoline should be added and has not been avoided, a jump to the equivalent address in the new (instrumented/optimized) code is inserted. This is done in `ConversionHelper`::*insertTrampolines()*.

There are 3 algorithms for placing the trampolines: *immediate*, *delayed* and *anywhere*. They all use a bitmap of the original code address space to mark the addresses that have been used to place jumps and which must therefore not be overwritten again.

63. This is only used when instrumenting, not for optimization.

64. Additional addresses may be specified using the `--trampolines-on-syms` command-line option.

The *immediate* mode is the most sophisticated method. It analyses all locations where a trampoline should be added and adds it directly if it cannot clash with other trampolines. At places, where only a shorter jump may be placed, because there are other trampoline locations in the way, only a short jump to a near full-length jump is inserted.

In some cases, the place with the trampoline can be left intact if there are no branches in the way – the trampoline can be safely deferred a couple of instructions later. If there is a `RET` instruction, no trampoline is needed, because the code will return to the caller anyway. If there is another trampoline, the two may use only one jump together, with one path running a little longer in the original code.

When using the *delayed* mode, basic blocks are used. All blocks that should have a trampoline inserted are pushed in a queue. Blocks from the queue are processed, one by one. If there is enough space, a trampoline is inserted, overwriting the block. Otherwise, the next block and the branch block (if there is a branch) are added to the queue and processed later.

In the *anywhere* mode, trampolines are placed in every basic block where a full-length trampoline jump may fit.

4.2.3 Other information transfer

To enhance debugging in the instrumented or optimized sections, some debugging information in the DWARF2 format, such as line info, is created for the new code. The information is taken from the original code and is translated using VMA translation.[65]

There is a plan to create exception information for the new code, but it is not implemented yet. The exception handler uses stack frame information similar to DWARF2 format frame info, but stored in the `.eh_frame` section, to unwind the stack and find appropriate handler for an exception. Once this work is done, exceptions should work even in the instrumented and optimized programs.

When all information is transferred, the start address of the program is updated to point to the new code.

4.2.4 Creating the output file

Just before writing the new code to the instrumented/optimized program, `BasicBlock` objects are destroyed and their raw data is written to data buffer in `ProgramSection`::*prepareSectionData()*.

The raw data from all sections, symbols, relocations and various flags are fed to the BFD library, which produces the result file.

65. The VMA translation, implemented in `CodeTracker`, translated between original and new virtual memory addresses (VMA).

Sometimes,[66] it produces a program that could not start. The most obvious reason is that it doesn't create an appropriate `LOAD` segment for some of the sections. One reason for this is that it cannot move the ELF program header table, which is usually at the beginning of the file, to another location. The header therefore cannot grow and new segments cannot be added to it.

The code in `Elf32_utils`::*fixFile()* goes through all sections from the section header table and determines if there is a `LOAD` segment with appropriate permissions which fully covers the section. If not, an existing `LOAD` segment must be extended or a new segment must be added to cover the section.

A new section `.elf-prgm-headers`, which serves as a placeholder for a new file header, is inserted into the program at the beginning of the optimization process. If the new program header table would not fit in the original space, it is written to this new section and a pointer in the ELF header is updated to reference it.

4.3 Instrumenting

Instrumenting is the process of adding profiling-gathering code into a program. This code provides profiling information to the optimizer. Currently, the profiling data contains the number of passes through all basic blocks and the number of passes through taken branches in all blocks. All control flow on edges, except on branches that jump to multiple targets (using indirect branches), is therefore known.

In the future, additional information, such as counting the times a register or memory variable is 0 or counting passes for indirect jumps, may be added.

4.3.1 Inserting counters

At the entry of every basic block, a counter is placed. When the block doesn't end with an unconditional jump, another counter is placed at the end of the block. This way, the number of straight passes through the block and branches can be calculated.

For indirect branches, which have multiple branch targets, the number of passes is estimated. The total number of taken branches from a particular block is divided – either equally or in proportion to target blocks input pass counts – to all branch targets.

The code for a single counter for the IA-32 architecture is written in the assembly language (source code is in `instrumenting/helpers-i386/counting_bits.s`) for a complete control over the code:

```
    PUSH ECX
    MOV  ECX, counterLO
    LOOP 1
    PUSHF
    LOCK
    INCL counterHI
    POPF
1:  MOV  counterLO, ECX
    POP  ECX
```

66. Most of the time, actually.

The counter consists of two 32-bit values: `counterLO` and `counterHI`. It needs to increment quickly the lower value, preferably without changing flags.[67] The `LOOP` instruction can be used for this on all processors. The only thing is that it decrements the value, instead of incrementing it. The lower part of the counter must therefore be inverted before use. The high part of the value is seldom incremented, therefore all flags are pushed and the `INC` instruction is used.

The profiling mechanism should not interfere with the other code. Even though stack is used to store temporary values, this should not affect any well-written program. The reason is that interrupts may normally arrive at any time during program execution and they may change anything below the stack pointer. The profiling code only uses memory that must not be used anyway, because of interrupts.

There can be another problem with the profiling code: concurrent access. This is because static variables `counterLO` and `counterHI` are used. Updating these values may not be atomic when running in multi-threaded environment. For the sake of limiting the impact on performance, the lower part – `counterLO` – is not protected, because writes are atomic and small errors in the value are not a big problem. The higher part – `counterHI` – is protected by a `LOCK` instruction, because the overhead is lower (due to lower update frequency) and a change in the most significant part could alter the interpretation of profiling data significantly.

On the x64 architecture, the situation is a little bit more difficult, because programs may use up to 128 bytes below the stack pointer as a scratch space. The stack pointer has to be lowered first[68] and then, data may be written below the original boundary.

Symbols that marked a basic block start in the original code, are placed on the counting code at the entry of the corresponding basic block in the instrumented code. Another symbol, with "`-direct`" suffix appended, is placed just after the counting code.

Information about every counter is stored in the `.info.ia32binopt` section of the instrumented program. Other info, such as the framework version, number of blocks and counters or section sizes, is placed in the `.data.ia32binopt` section. Data from both sections are later used for analysing the counters in the optimizer.

4.3.2 Code helpers

There are several tasks that need to be done before counters can be used. Therefore, an initialization code is injected into the code and the entry point of the program is redirected to it.

The initialization code is fairly complicated to be written in pure assembler, so it is split in two parts. Most of the code is written in C, only a short routine is coded in assembler.

67. Saving and restoring flags is an expensive operation on modern processors.

68. This has to be done to protect the data of the profiling code against interrupts, which may overwrite anything below the 128-byte protected area.

The initialization assembler code, which is run before any other code of the program, is located in `instrumenting/helpers-i386/counting_bits.s` and looks like this:

```
PUSH EAX
MOV  EAX, [ESP + 4]
MOV  program_argc_ia32bo, EAX
LEA  EAX, [ESP + 4*EAX + 12]
MOV  program_envp_ia32bo, EAX
LEA  EAX, [ESP + 8]
MOV  program_argv_ia32bo, EAX
PUSH ECX
PUSH EDX
CALL initializeCounting
POP  EDX
POP  ECX
POP  EAX
JMP  origStartVMA
```

It fills some important variables which will be used later and calls the C initialization routine in *initializeCounting()*. At the end, it jumps to the original entry point.

The part of code written in C needs to do a lot more – prepare file to store the counter values and make changes to the memory layout.

The path to the counter file is determined by appending the absolute path of the program after the root directory for counter files. The root directory for all counter files is normally either `/tmp/counters` or – if the environment variable is defined – `${IA32BINOPT_BASE}`. This location assures that the instrumented file doesn't need any special permissions for updating the counter file.

The initialization code has to create the counting file if it doesn't exist,[69] and erase its contents if it describes an older version of the program.[70] The directories are created with special permissions, similar to those usually used by `/tmp`, which ensures that everybody can write to their own counters, but not to anybody else's.

If the counter file cannot be created or loaded, the area used by the counting code is allocated and cleared. The code will work normally, only values of the counters for that particular execution of the program will be lost.

The counter values are preserved and reused across different invocations of the program. To achieve this, counters are mapped from a file using *mmap()* system call, which automatically updates the values on program exit and on other occasions.

A problem might occur when two different instances of the same program would be loaded into memory. This is resolved by using an optional locking of the counter file. If the locking fails, which means that another instance of the same program is

69. All directories on the path must also be created if they don't exist.

70. This is detected using the `.info.ia32binopt` section, which contains information about the program and also describes all counters.

running, the initialization code must *mmap()* anonymous memory instead of the file, because the counting code will be writing to the area in all cases.

Some systems also incorrectly set the *brk* value – which marks the end of the data segment and start of an area available to heap – to a low value. This can cause a collision between the counter area and heap. The initialization code therefore checks the *brk* value and updates it if necessary.

The instrumented program doesn't necessarily dynamically link to the standard C library or may import only a couple of functions. It would be therefore difficult to use standard functions in the C initialization code. To allow the initialization code to use a C library, it needs to be statically linked. The standard *glibc* library was evaluated, but it cannot be used two times, both statically and dynamically linked, in one program. The solution was to use the *dietlibc* library, which provides all basic functions and is very light-weight. The initialization self-contained code is roughly 2.5kB, including the parts of *dietlibc* it uses.

4.3.3 Helper object file

Embedding the initialization code or the code of the counters into the program that performs the instrumentation would limit flexibility. This code is therefore read from a relocatable object file, which is located in `helpers-i386/counting_bits.o`, relative to `ia32bopt_prepare` program path.[71] This default location may be overridden using the `--code-utils-file` command line option.

The counter code and the initialization code are placed into separate sections, called `.text.cntblock` and `.text`, respectively.

The counting code is placed into a `BasicBlock` object and then cloned to all places where a counter is needed. It uses two relocations on IA-32 – for the `counterLO` and `counterHI` symbols. For every occurrence of these symbols, a `Relocation` object is created and the location is relocated before the section data is written to the destination file.

The initialization code is copied (in *fillHelperSection()* in `prepare_counting.cpp`) into a new section `.text.ia32binopt`. All data used by this code are copied into new sections `.data.ia32binopt` and `.bss.ia32binopt`. Symbols and relocations are copied from the helper object file, too.

Some undefined symbols from the helper file, such as `origStartVMA`, `counters`, `counters_count` or `program_argv_ia32bo` are defined, allocated space and pointed at their final place in the new sections. The `errno` symbol needs a special attention. In the helper object file, there is a relocation for this symbol pointing at the first byte of the `.bss` section. In the instrumented file, this location is occupied by information about counters, so a new place for errno is created in the `.data.ia32binopt` section.

71. This applies to the IA-32 architecture. Code for the counters and the initialization rountine for the x64 architecture is stored in `helpers-AA64/counting_bits.o`.

4.4 Analysis

Before the program can be optimized, it is first analysed. The routine which conducts the analysis is *analyseInvariants()* in `optimize.cpp`.

First, a control-flow graph (CFG) graph and reverse CFG are built for all blocks in the code section. An `InstructionInstanceWI` object, which contains data structures for storing invariants, is created for all instructions.

4.4.1 Analysis of the stack pointer

Stack pointer value relative to a base value is recorded for all instructions. This is done for all `Function` objects. The value of the stack pointer at the entry of the function, or at the first block – in case there are multiple entries – is used as the base value. All operations with the ESP register are recorded and the stack pointer position relative to the base value is updated. A check if all stack positions match the stack position of target block in cases of a jump, and also if the stack position at a return from a function is the same as at the entry.

4.4.2 Analysing free locations

After stack pointer analysis is done, free space in registers and on stack is explored using an iterative algorithm.

In the beginning, all blocks are added into a queue and all sets of empty (or free) locations are cleared. Blocks are then taken from the queue, and their free locations are analysed.

For each block, empty locations from the beginning of all subsequent blocks in the control flow are taken and their intersection is used as the state of the empty locations at the end of the block. The instructions in the block are then processed backwards, adding all locations that are overwritten to the empty locations set.

When the empty locations at the beginning of the block change, all predecessors in the control flow graph are added to the queue to be processed again, because the initial set of empty locations at the end of the block may have changed.

All changes in the set of empty locations are monotone, new empty locations are only added, never removed. The algorithm is finite, and can be stopped at any time, yielding a valid set of empty locations, even though it may not be the largest one. There is currently no limit on the number of iterations, as the algorithm tends to converge quite fast.

The `OptimizeFramework::analyseEmptyLocations()` contains the code for the analysis.

4.5 SSA Form

The *static single assignment* form (SSA) is an intermediate code representation often used in compilers. In this form, variables are versioned and every version of a variable is assigned to exactly once. When a variable is assigned a different value, a new index is chosen for it and for all subsequent usages of it. When two different versions of a variable come to a basic block from two different control flow branches, a new version is created. The new version is assigned the result of a ϕ-function with all the input versions as parameters.

```
x = 2;                          x1 = 2;
y = 3;                          y1 = 3;
z = 5;                          z1 = 5;
if (...) {                      if (...) {
   x = x + 5;                      x2 = x1 + 5;
} else {                        } else {
   x = x + y + z;                  x3 = x1 + y1 + z1;
   y = 2 * y + x;                  y2 = 2 * y1 + x3;
}                               }
                                x4 = φ(x2, x3);
                                y3 = φ(y1, y2);
x = 2 * x + y + z;              x5 = 2 * x4 + y3 + z1;
```

Figure 2. An example of translating a program to SSA form

The SSA form has the advantage that many properties of the data flow become evident. For every usage of a variable, it is easy to track its definition and all other usages of the same value.

For a binary framework, all usages of registers, and stack locations will be assigned an index.[72] This index will be kept in the `InstructionInstanceWI` object, which represents the instruction. Additionally, a database mapping from indexed variables to their definitions and all usages will be created, if necessary.

The SSA form is constructed in several phases, discussed in subsequent sections. The source code for SSA form building is in `SSAForm`::*buildSSA()*. It is almost finished, except for the last step – the actual construction of the form. No optimizer plugins can take advantage of it yet.

More information about SSA can be found in [37] and [38]. Dominators and dominance frontier construction are analysed in [39], [40], [41], [42], [43] and [44], data structures used in building of the dominator tree are described in [45], [46] and [47].

4.5.1 Dominator tree

The *dominator* and *post-dominator* are important constructs in the control flow. They are used in the SSA construction, but can have many other uses in control flow and data flow analysis.

72. The SSA analysis is usually done inside one function, so stack locations are stable. If building global SSA form, indexes for stack locations would have to be built separate for different functions and then united for each analysis across calls, considering stack base differences.

A node through which all possible control-flow paths from entry to node x must go, is called a *dominator* of x. Similarly, a node through which must go all paths from x to the exit node is called a *post-dominator* of x. The *immediate dominator* of a node x is the last dominator of x on any path from entry to x. It can be shown that it is unique. The *immediate post-dominator* is defined accordingly. The immediate dominators form a tree called a *dominator tree*.

In the optimizer framework, dominator trees are built separately for each function. Nodes in the definition are represented by basic blocks and the entry node is the block with the entry to the function. Any basic block with a `RET` instruction is an exit node.

To evaluate different algorithms, four methods can be used to build a dominator tree: Lengauer-Tarjan [39], Semi-NCA [43], iterative DFS [42] and iterative BFS [43].

4.5.2 Building the SSA form, J-reduced CFG, ω-DF

The *dominance frontier* of a node (basic block) x, denoted DF(x), is the set of nodes that are not dominated by x, but some of their immediate predecessors in the CFG are dominated by x. This means that the nodes in the dominance frontier of x are nodes that have a path leading to them from x and also a different path from the entry node, which does not pass through x. The nodes in the dominance frontier of x are candidates for placement of a ϕ-function for all variables modified in the block x. Building a complete DF may be too expensive, as suggested in [37], so the ϕ placement algorithm doesn't build the DF explicitly.

To build the ϕ-function, a J-reduced CFG is built, which collapses all strongly connected components of the dominance frontier graph into a single node. These components are the same as the strongly connected components of the ω-DF graph, which is a restriction of DF on siblings in the dominator tree and is smaller than the complete DF graph.

When a J-reduced CFG is constructed, siblings in its dominator tree are ordered according to a topological order of the ω-DF graph. The post-order visit to the dominator tree of the J-reduced CFG gives a an ordering called ω-ordering and it can be shown that it also constitutes a topological sorting of the dominance frontier graph.

The nodes are then processed in the ω-order and for each node a pruned dominance frontier (PDF) is built, using information from its children in the dominator tree[73]. Nodes in the PDF(v) set of block v will inherit variable versions assigned in v and also versions from other blocks, so a ϕ-function must be placed there for all variables assigned in v.

We only need a single pass, because the ω-ordering is also a topological sorting of the dominance frontier relation, so dominator tree children are always processed before their parents and also blocks in the DF relation are processed in the right order.

73. Because the ω-ordering is a post-order ordering in the dominator tree, all children must have been processed before their parent.

4.6 Optimization

When the analysis of the code is finished, the program is ready to be optimized. All optimizer plugins that are included in the build are instantiated and registered at an `OptimizePluginRegistry` object. It reads a configuration file, updates values in plugin configurations and executes the plugins. All plugins inherit from the base class `OptimizePlugin`, which provides the interface and configuration handling.

Every optimizer plugin gets an `OptimizeContext` and `OptimizeFramework` objects which contain analysis info about the code, such as counters, empty locations, control flow graph, dominator tree and in the future also the SSA form.

Most plugins work only on "*hot*", or frequently executed, code. This is usually set as a relative value, using a percentage of the most frequently executed block in the program.

4.6.1 CacheUnalias plugin

The *CacheUnalias* plugin[74] aims at improving instruction cache performance by moving or copying basic blocks to different addresses. This plugin yields the biggest performance gains, as experimental results in *Chapter 5* suggest.

Instruction cache performance can be limited by two problems: address aliasing and partial cache line utilization.

When several addresses contend for the same set in the instruction cache, they keep pushing each other out from the cache. This is called *address aliasing* and can impact performance, especially when the aliased addresses are inside a tight loop. This can easily happen for example when a code in a tight loop calls a function placed at a distance approximately equal to a multiple of the cache size.[75]

Another problem is called partial cache line utilization. Code is always cached in contiguous blocks, usually 64 bytes long. When the frequently executed code is scattered around memory, lot of other unused (or less used) memory can be cached with it.

Both problems can be solved by improving code locality – compacting frequently executed blocks together. The partial cache line utilization is improved by moving frequently used blocks to fill cache lines, address aliasing is minimal when addresses are sequential. The algorithm for cache addressing usually takes the address modulo cache line modulo number of sets as the set index. This way, when the frequently executed blocks are placed next to each other and they fit into the cache, there is no aliasing.

The algorithm works by going through basic blocks. It ignores all blocks that are not in a *hot-spot* – that have a lower pass count[76] than a specified percentage of the highest pass count.[77]

74. Source code of the CacheUnalias plugin is in `optimizer/plugins/CacheUnaliasPlugin.[h|cpp]`.

75. It will happen more precisely when the distance is a multiple of n, which is the cache size modulo the number of ways of the cache.

76. The number of passes through the block, determined from the profiling data.

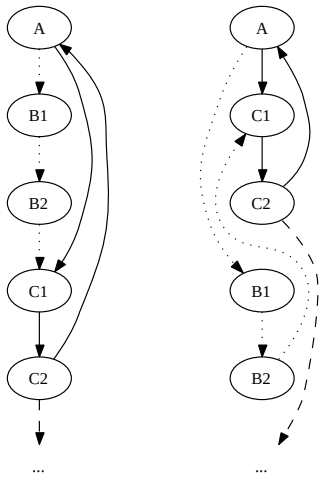
77. This setting is controlled by the `percentTreshold` setting in the config file.

For other blocks, it tries to put the most frequent ("*hottest*") path straight. This means that if the most frequent path is using a branch, then the branch target block (the most frequent of all target blocks) is moved just after the current block and searching continues from that block.

This way, frequent paths are linearized and hot-spots are compacted. Normally, branches that go to another already processed hot-spot are ignored. When the config option `mergeHotSpots` is enabled, a hot-spot can be merged with another one.

In some cases, a block is used from several places. This can be the case of a shared function entry point. It can be moved next to only one block. If a block is used extensively from many places, it can be copied next to each usage. This is controlled by config options `enablePartialInlining` and `partialInlineTreshold`. The first one enables or disables block copying, the second one specifies (in percents) the maximal ratio of passes into the target block that come from the current block to copy the target block next to the current block instead of moving it.

If there is a larger percentage of passes coming from the current block, the target block is moved next to the current block. If there are less, the block is only copied. If there are fewer passes from current block to the branch block than to the next block, the branch target block is not moved and search continues with the next block.




```
A:    JNE C1          A:    JE  B1
B1:   ADD EBX,7       C1:   CMP EAX,5
B2:   XOR EAX,EBX     C2:   JGE A
                            JMP out
C1:   CMP EAX,5       B1:   ADD EBX,7
C2:   JGE A           B2:   XOR EAX,EBX
                            JMP C1
out:                  out:
```

Figure 3. The effect of the *CacheUnalias* plugin

The *Figure 3* shows a CFG graph and assembler listing of an optimization change done by the *CacheUnalias* plugin. The ellipses are basic blocks, solid lines represent frequently executed paths. The *hot* loop $A \rightarrow C1 \rightarrow C2$ is linearized, at the expense of some additional jumps. Note, that when the successors of a block ending with a conditional jump are changed, the condition must be inverted.

While the *CacheUnalias* plugin works on basic blocks, the similar *HotColdSeparate* plugin operates on whole functions.

More information about improving code locality can be found in [48], [49], for more information about code positioning in general, see [50], [51].

4.6.2 BranchAlign plugin

Some processors always fetch aligned 16-byte blocks to the instruction decoder. When a jump target is not aligned to a 16-byte boundary, a part of the first fetch block after a jump is unused. This plugin aligns the most frequent jump targets to an address which is a multiple of 16.[78]

Only blocks, that have at least a certain frequency and a much larger input flow from branches than from normal control flow are considered.

If blocks that are not frequently executed were aligned, it could result in considerable code size increase without much gain. The performance might probably even be hurt because of cache performance.

If padding was inserted before a block with a significant normal (non-branch) inflow, the performance would be also degraded – both because of cache issues and because padding instructions also take some time to decode.

```
0x17  MOV EAX,4          0x17  MOV EAX,4
                         0x1a  LEA EDI,[EDI + 0]
0x1a  ADD ESI,1          0x20  ADD ESI,1
...                      ...
0x31  JNE 0x1a           0x37  JNE 0x20
```

Figure 4. The effect of the *BranchAlign* plugin

The *Figure 4* shows a snippet of code before and after the optimization, with instruction addresses in the left column. The `LEA` instruction is used as a `NOP` and aligns the `ADD ESI,1` instruction, which is a target of a frequent jump from `0x31`, to an address aligned to a multiple of 16.

The padding instructions for alignment are not inserted into basic blocks in this plugin, because the final addresses are not known yet and may still be changed by other plugins. The alignment requirements are only marked with the block. Padding is inserted when blocks are positioned in `ProgramSection`::*condenseBlocks()*.

The `globalPercentTreshold` option specifies the minimal pass count of the block to be considered for optimization. It is a relative value specified as a percentage of the maximal pass count of all blocks in the program.

To align a target of a jump, the inflow from jumps must be at least n times larger then from normal control flow. The n is specified using the `jumpAlignTreshold` option.

More information about branch aligning can be found in [27].

78. Alignment to 16 is the default value. It can be changed using the `alignSize` config option.

4.6.3 AthlonBTB plugin

On Athlon processors, the `RET` sometimes doesn't load a BTB entry or aliases with another branch in the BTB table.[79] Both problems can be avoided if the `RET` instruction uses a longer encoding, which can be achieved using a `REPZ` prefix (`0xf3`).

This is done only in the most frequently executed parts of the code, because it makes the code longer.

```
0x1c  39 c8  CMP EAX,EAX      0x1c  39 c0  CMP EAX,EAX
0x1e  74 01  JE ADD ESI,1     0x1e  74 01  JE ADD ESI,1
0x20  e2 fa  LOOP 0x1c        0x20  e2 fa  LOOP 0x1c
0x21  c3     RET              0x21  f3 c3  RET
```

Figure 5. The effect of the *AthlonBTB* plugin

Results of the optimization are shown in *Figure 5*, with addresses in the left column, instruction encoding bytes in the middle and instruction mnemonics on the right. A `REPZ` (`0xf3`) prefix, which has no effect on the `RET` instruction, is added to increase the instruction encoding size.

More information about Athlon BTB can be found in [25].

4.6.4 HotColdSeparate plugin

The *HotColdSeparate* plugin is similar to the *CacheUnalias* plugin when partial inlining is not used, but it moves whole functions instead of basic blocks.

Functions that are used frequently will be grouped together and also functions that call each other will be kept close.

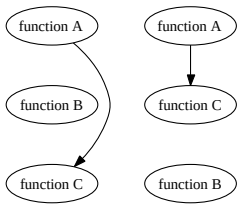


Figure 6. The effect of the *HotColdSeparate* plugin

Results of the optimization are shown in *Figure 6*. The function C, which is frequently called from A, is moved next to A. The function B is shifted to make place for C.

This plugin is marked experimental.

79. For more information, see the Athlon description in Chapter 2.

4.6.5 FunctionInline plugin

The *FunctionInline* plugin is similar to the *CacheUnalias* plugin with partial inlining, but it copies whole functions instead of basic blocks. Functions that are used frequently will be grouped together and also functions that call each other will be kept close.

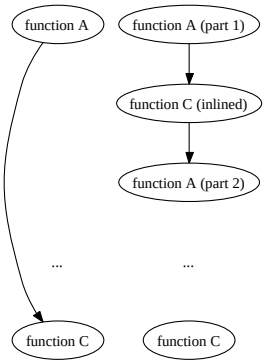


Figure 7. The effect of the *FunctionInline* plugin

Results of the optimization are shown in *Figure 7*. The function C, which is frequently called from A, is moved inside A, at the place, from which it is invoked. The function B is shifted to make place for C.

The `hotspotTreshold` option controls the area where the optimization will be done, `maxInlinedSize` specifies the maximum size of a function that may be inlined. The `moveTreshold` option is the necessary treshold to use code moving instead of copying for inlining. If a caller of a function produces more than `moveTreshold` percents of passes into the function, the code will be moved. Otherwise, the code of the function will only be copied.

This plugin is marked experimental.

4.6.6 DeadCodeRemove plugin

This plugin eliminates unreachable code. It works by removing functions that are not referenced from any other code or data. The reachability analysis is repeated several times, until no more functions can be removed. This iterative approach is necessary to remove code that is referenced only by other dead code.

The effect should be a reduction of total code size and may also result in improved cache locality.

This plugin is not implemented yet.

5 Experimental results

To determine the efficiency of optimization, a number of benchmark experiments have been conducted on different processor architectures.

Performance of optimized programs have been measured with various optimization options. The results are summarized in the following section.

All benchmark scripts and raw benchmark data are included on the CD.

5.1 Benchmark measurement

To provide the same execution environment for all tests, a Linux live CD based on Gentoo – *Kororaa* version 0.2 was selected, because it was supported on all target machines.

All systems were booted from the CD with a boot option `softlevel=nox`, to start to a clean environment with as few running processes as possible.

When the system is booted, an USB flash disk with the benchmark suite has been inserted, mounted and the `run_benchmarks.sh` script on it was started.

This script first copies all tests into a ram-disk, compiles the framework and other required libraries and runs the tests. The actual tests are performed by a perl script stored in `scripts/benchmark.pl`.

5.1.1 Tests

Each test measures the optimization impact on one program. It starts by instrumenting the program and running the instrumented version. This provides profiling information for optimization and – as a by-product – it should bring all input data into memory.

After that, the timing starts. The execution length of the original file is measured first, followed by that of the instrumented file and the optimized files.

To measure the impact of various optimization options, a program is optimized and timed with different configurations. The configuration file for the optimizer is changed before each measurement.

There is a list of all values for one configuration option that should be tested. All combinations, where only one optimization plugin is active at a time, are measured. The restriction to only one active plugin was added because of the large number of combinations. Even with these restrictions, the number of individual timings is 160 or 120, depending on the test. Each such timing consists of 35 executions of the tested program. In later tests, the number of timing cases was reduced – configuration option combinations which did not yield good results were discarded.

5.1.2 Timing

The `Time::HiRes` perl library was used for measurement. Current time was taken before and after execution of the test program and the difference was taken as a result.

To suppress random disturbances, the program was run 35 times and worst results were discarded. They were regarded as deviations possibly caused by scheduling and

interrupt events. These discarded results should also cover the first execution, which brings all important files into memory and may therefore took longer (even though tests were run from a ram-disk). From the remaining 25 executions, the mean value was taken as the final result of the timing.

5.1.3 Various test programs

Several programs with various test data were analysed. They were selected to represent diversified test samples.

Compression programs are represented by `bzip2`, `gzip` and `rar`. They were run on a compressed file to test its integrity. The size of the file was about 10.5MB for `gzip`, 9.5 MB for `rar` and 4.2 MB for `bzip2`. A bigger test file was selected for `rar` to measure the impact of longer running time on the optimization level.

Another group of tested programs were language interpreters: `perl`, `php` and `python`. They were called on sample scripts, written specifically to test different aspects of the interpreter. For `perl` and `php`, two different scripts were used.

The GNU `gcc` compiler was tested with compilation of a simple program. It may have spent more time in various initialization routines than doing the actual compilation, because the compiled file had just about 270 lines.

As a result, the hot-spot was not clear and optimization didn't work very well for `gcc`, as will be shown later.

The web browser `links` was tested to render a 5 MB HTML page with tables. It was selected, because unlike the compression programs, it doesn't have a single compact hot-spot and many calls may be used in the inner loops.

The last tested program was `dcraw`, which converts raw data from digital cameras. It was run on a 13.4 MB large raw picture from a camera. This program differs from the others in that it was compiled from a single source file, so the compiler could perform all optimizations in one compilation unit. The test was also by a wide margin the longest running test.

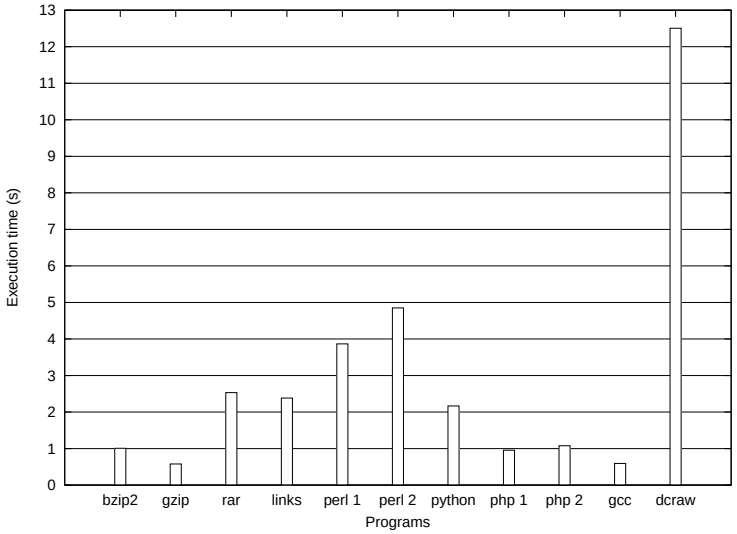


Figure 8. Total execution time of different test programs.

The *Figure 8* shows average execution times of the tests. The execution time in many cases influences how important the hot-spot will be, relatively to other parts of the code and to program load time.

All files optimized by the framework have more code segments and longer code to load into the memory. Many memory pages in the original code are also loaded from file when trampolines are frequently used. There is therefore an inherent penalty.

Because of time constraints, the `gcc` and `dcraw` tests were not performed on all processors.

The `bzip2`, `gzip` and `rar` were taken unmodified from an installed system, other programs were configured and compiled on the test machine.

5.1.4 Test machine configuration

There were four processors tested:

- AMD Athlon XP 2500+ (1830 MHz)
- Intel Mobile Celeron 2.0 GHz (based on Pentium 4), scaled at 1 GHz
- Intel Pentium M 740 (1733 MHz)
- Intel Core Duo T2300 (1666 MHz)

The Celeron/Pentium 4, had $1 \cdot 512$ MB. Athlon worked with $2 \cdot 512$ MB dual-channel (at 400 MHz) DDR memory, Pentium 4 had $2 \cdot 256$ MB (at 266 MHz) DDR memory, Pentium M used $2 \cdot 512$ MB dual-channel (at 533 MHz) DDR2 and Core Duo had $1 \cdot 1024$ MB DDR2 (at 666 MHz).

The Mobile Celeron, based on Pentium 4, had significant heating problems, so it was scaled to work at 1 GHz.

5.2 Results

The optimized programs have an inherent penalty associated with the use of trampolines. If a program uses many indirect jumps, it may perform significantly worse than the original code.

5.2.1 Instrumented versions

While not critical to the success of an optimization framework, it is nevertheless important to keep the overhead of the instrumentation as low as possible.

On Athlon, the instrumented version ran about 2.5 times slower for most programs, 3 times for `php` and `python` and 4.5 times slower for `perl`, on Pentium 4, it ran only about 1.75 to 4.25 times slower, on Pentium M, it was about 2.2 to 4.5 times slower.

5.2.2 Optimized versions

The optimized versions performed reasonably well for some programs, even though not on all processors. Because of time constraints, I wasn't able to finish measurements of `gcc` and `dcraw` on some processors.

The results of timings of optimized versions are presented in the table below:

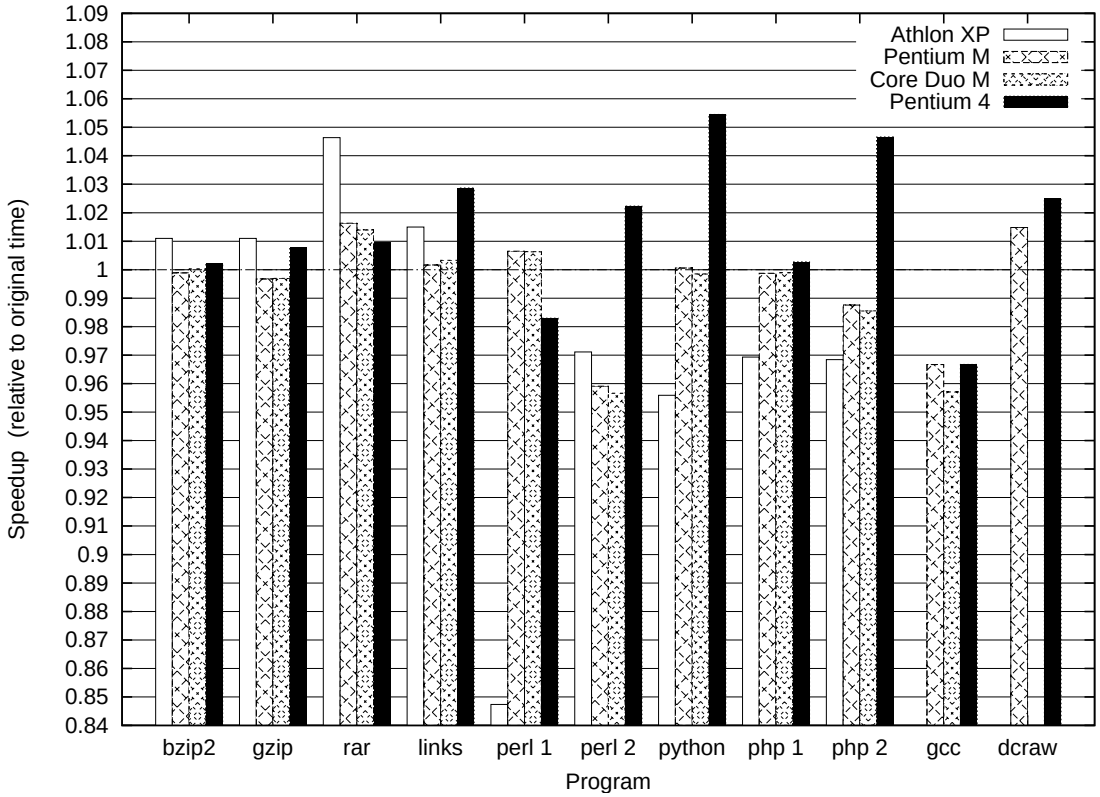


Figure 9. Optimization impact with various test programs, relative to original time.

The result of the optimization with the optimizer configuration, which provided the best results, was taken and compared with the original file. *Figure 9* shows the ratio, the original time divided by the time of optimized program.

We can see that for certain programs, such as `rar`, the optimized program worked clearly faster on all configurations. This program has a hot-spot split into several parts, and can benefit from joining them.

On the other hand, some programs, most notably `gcc`, performed worse on all processors. This can be explained by short running time and possibly using many indirect jumps, which causes taking many trampolines back to the optimized code.

An interesting case is the `perl` program with the first data set, which manifested the worse result – it scored 15% below the original program on Athlon! I am not able to fully explain this drop in performance. The Pentium M and Core processors were able to outperform the original program on the same data, which probably means, that excessive using of trampolines cannot be blamed for the result.

It should be also be noted that optimized versions performed significantly worse on the Athlon for all interpreters: `perl`, `python` and `php`. The optimizer was not able to reach even the performance of the original programs. It might be partly blamed on using a lot of indirect branches, but there has to be a reason, why the other processors performed much better.

5.2.3 Impact of optimization parameters

Another interesting thing is to measure the correlation between optimization parameters and performance.

The `rar` program, which optimizes reasonably well on all processors, was selected for the measurement.

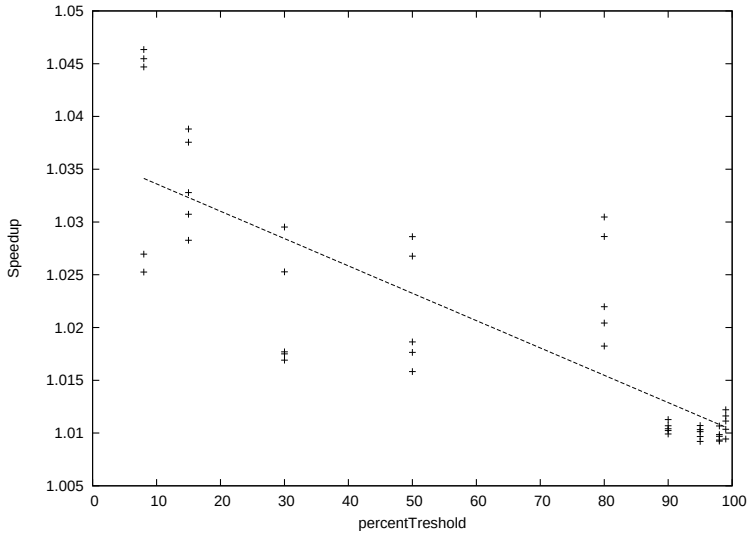


Figure 10. Correlation between optimization parameter `percentTreshold` and speedup.

The strongest correlation was measured between performance and decreasing value of the `percentTreshold` option of *CacheUnalias* plugin with the `mergeHotSpots` option enabled. Another options with strong correlation to performance increase were settings concerning partial inlining.

The `percentTreshold` option selects the working set or hot-spot. The smaller is the value, the larger is the area that is optimized. Measurements were taken from the Athlon benchmark measurement results with `percentTreshold` values 8, 15, 30, 50, 80, 90, 95, 98, 99.

Only measurements using *CacheUnalias* plugin with with `mergeHotSpots` option enabled were taken into account. Every dot represents a measurement performed with different settings of partial inlining.

The graph shows that for low values of `percentTreshold`, other options play an important role. Especially the `partialInlineTreshold` value. For most lower values of `percentTreshold`, the dots are divided into two groups. The 3 higher ones usually don't use partial inlining or use a low `partialInlineTreshold` value of 40 or 65, lower dots use 90 and 95. When only the innermost loop is optimized, this option doesn't make such differences, because partial inlining is performed only on the hot-spot.

6 Future work

Although some encouraging results were obtained from the optimizer framework, there are many open possibilities for improvement. Some features are not finished yet, such as the SSA analysis and optimizations based on that form. In this chapter, some of the possible future directions are explored.

6.1 SSA form

Construction of the SSA form is complicated, but brings a whole range of new opportunities for optimization. The optimizer plugins will be able to tell what values may a variable[80] contain or with what other variables it is aliased.

This can be useful for example to eliminate a check of a value for another value, when it can be deduced that the test will always be positive or always negative. Code with these properties is often generated by functions that check their arguments for incorrect values, for example for a NULL. If the value passed to the function is correctly checked in all callers, it is useless to check it again.

Closely connected to this is dead code elimination of unreachable code. If a test condition of a branch always has the same output, the other branch is never executed and can be removed along with the check.

Note, that a compiler cannot perform such checks for non-static functions, because it does not know all possible users of a function.

The SSA form will also make it possible to easily find an assignment to a variable for a particular usage or find other usages. This can help, for example, to find possible jump targets to improve control flow analysis.

6.2 Support for exception handling

Exception handling raises a couple of problems with binary code transformations. It can be implemented in different, mutually incompatible, ways. If the code conversion does not understand exceptions, the resulting program will almost certainly crash on the first exception thrown in the converted code.

When exceptions are raised, a function usually called *__cxa_throw()* is called. It searches all stack frames on stack for a handler of the exception. To find the handler, there must be additional exception frame information, usually stored in the `.eh_frame` section in the ELF file. This section is mapped to memory, so that the routine which throws exceptions can use it. The stack frames are searched from the newest to the oldest until a handler is found. For each stack frame, the return address is used to search the exception frame information. The address of the next frame may also be difficult to calculate, so it is taken from the frame info as well.

80. By a variable, I mean either a register or a memory location on the stack.

When there are no records about stack frames and exception handlers for a particular address, the handler for the exception will not be found and the program is terminated.

The convertor should generate exception handling information for the new code section, based on the information for the original code. When this is done, programs with exceptions will work much better.

An alternative approach for exceptions is to use *setjmp()* and *longjmp()*. This method should work better with converted code, because no hard-coded table of addresses is used, but this mechanism has been superseded by other methods and is not used very often any more. Other problems with unexpected control flow may arise, though.

Note, that even programs that use exceptions may work flawlessly if no exception is thrown. This is the case of the RAR program, which works well both instrumented and optimized, unless an exception is thrown. It is normally thrown only if an unrecoverable error in the file is encountered or when supplying incorrect command-line parameters.

6.3 Support for the x86-64/x64 architecture

Currently, the framework only supports the x86 (or IA-32) architecture. The x64 architecture is similar and is getting more popular, so it would be useful to add support for it. It has several notable differences – the ABI specifies that there is a scratch space below the stack, where data must be preserved. This presents a challenge for counters, because they must move the stack pointer and then write to the area that just became the new scratch space. The code for this is written but not tested yet.

Another problem is with RIP-relative addressing. This means that to understand the instruction data, its original location must be known. In the current implementation, it is possible to query the original address for code that came from the original file, but not for other code. The translation routines are also not meant to be called for every instruction.

To solve this problem, some information must be added to every instruction. The `InstructionInstance` class can be used for that. All basic blocks will then contain a list of such objects.

For complete support of x64, some architecture-specific code dealing with instruction decoding and modification must also be completed.

6.4 Using processor-specific performance counters

Processors offer a wide range of data about execution of code. Cache miss rates, branch mispredictions, cache alias conflicts or pipeline flush events can usually be observed using architecture-dependent performance counters. They could provide an important increase in profiling data gathering and help to identify the bottlenecks of the code.

6.5 Improved control flow analysis

To achieve good optimization level, a correct control flow analysis, is necessary.[81] Direct jumps are well analysed and do not pose a problem. Some indirect branches are practically impossible to predict, but a large group of indirect branches can be reasonably well analysed.

Indirect branches are often generated for longer *switch* statements. They often use a simple jump table, where its start and end can be determined using instruction invariants, obtained for example from the SSA analysis. It is sometimes even simpler, because many compilers generate checks for the bounds just before the indirect jump, in the preceding basic blocks. In this case, no complicated inter-block analysis needs to be done.

Another group of indirect branches stems from usage of virtual functions. They are called via a jump table called virtual method table (VMT). If a careful analysis is done, many locations of these tables can be found. Often, the ECX register points to the VMT or at least a part of it can be easily deduced from the value of the register. If all possible values of ECX at a certain point can be found,[82] the jump targets might be found, too.

6.6 On-line optimizations

The framework could be transformed into a general just-in-time (JIT) optimizer which might reorganize code on the fly, when usage patterns would change. The current analysis by instrumenting code would not be feasible because of the large performance overhead, but a statistic sampling method, with under 1% penalty could be used.[83]

The overhead of the code reorganization and of the sampling might be offset by improvements of the program performance. Of course, this approach could be only practical for long calculations, which would occasionally, such as once per two hours, change runtime behavior.

The framework could also be used to optimize dynamically generated code. Using an optimizer from a compiler would be in this case difficult, because it usually requires a larger framework and a different type of input data produced from programming language source code.

6.7 Optimization of dynamic libraries

Instrumenting and optimizing dynamic libraries can be useful in many cases. It is not currently supported, but it could actually be easier than working with executable program files, because dynamic libraries contain position independent code and include additional information about all references to code, as they need to be relocated to a different address if needed.

---

81. It is not required that all control flow paths are discovered for indirect jumps – but the more is known, the better will be the possibilities for optimization.

82. This may be possible with the SSA form available.

83. There is already a statistical profiling system for Linux called `oprofile`, which might be used.

Adding support for dynamic libraries should therefore not be too difficult.

It would be interesting to allow instrumenting and optimizing an executable program with all libraries it uses. Imported libraries specified in the ELF header would be replaced by instrumented/optimized versions. The *dlopen()* library call would be routed to an injected routine which would intercept the calls, instrument or optimize the requested library and link the modified version in place of the requested library.

6.8 Other optimizations

Additional optimizations of code may be implemented. Some are improvements of current optimizations, some are new. Many of them require a deeper analysis above what is currently implemented.

Some optimizations may be considered dangerous if they do not take into account that a value might be modified from an interrupt context, from another thread or process[84] or from the system. For values in registers, and under certain conditions also on the stack,[85] it can be safe to assume that the value cannot be accessed from a different context. In other cases, either a more careful analysis has to be performed or some values have to be left out of optimization.

If the value of the stack pointer ESP gets out of sight of the analysis, it must be considered that all memory addressed by it may be unexpectedly overwritten. If it only stays in, lets say, ESP and EBP, the optimizer may assume that no pointer references the stack space of the current function and therefore the values stored there cannot be modified except by the code of the function, which is under control.

6.8.1 Completely inline small functions

Small functions may be completely inlined and the `CALL` and `RET` instructions removed. This would require updating all stack references crossing the location of the return address on stack, because the return address would be removed from stack.

The current code compaction plugins do a part of this, but they don't eliminate the `CALL` and `RET` instructions, which are superfluous.

6.8.2 Dead code elimination

Code that can never be reached,[86] including whole procedures, can be removed from the program. This will reduce code size and potentially improve cache performance.

Some code, which is never referenced altogether, may be removed without further analysis. Other code is referenced, but only from other dead code or as one of the two control flow paths from a conditional branch, which jumps always in the other direction. This case requires further invariant analysis to find branches that always go in the same direction.

---

84. This can happen for shared memory.

85. In functions where the stack pointer is "leaked" to another place, it can used as a pointer and passed to any function or system call.

86. Such code is called dead code.

6.8.3 Instruction scheduling

Instructions may be reordered, either inside one basic block or across blocks, to improve performance of scheduling done by the processor. This has to observe dependency chains, because instructions are partially ordered by dependency chains and not respecting this will cause using or overwriting a wrong value.

6.8.4 Data flow optimization

Many optimizations can be done on data flow. Instructions that write values, which are never used, my be left out.[87] Long data transfers, such as moving data via stack, may be shortened and the data written directly to destination – in case the intermediate values are not used. Usages of certain variables might be replaced by their aliases[88] if it reduces dependency chains, improves memory access or prevents ROB stalls. Constant variables may be replaced by constants.

6.8.5 Instruction selection

Instruction selection can also significantly influence performance. Some instructions or instruction combinations are slow on certain processors and there is often possible to use alternative instructions. On some processors, a `PUSH` instruction may be replaced by a `MOV` + `ADD ESP,4`. On other processors, which have a stack engine, a change in opposite direction may improve performance.

Some simple branches, where one of the alternative control flow paths only updates some variable[89] may be replaced by `SETcc` or `CMOVcc` instruction(s).

The Pentium 4 processor offers a lot of possibilities for improving performance by substituting similar instructions. Selecting `ADD` + `JC` instead of the `ADC` instruction is one example.

Some optimizations are rather non-obvious. On the AMD K6, encoding the memory reference `[ESI]` as `[ESI+0]` with a zero displacement improves decoding bandwidth.

6.8.6 Additional code reordering

A more advanced procedure sorting, possibly using an algorithm such, as in Pettis and Hansen [51] or the Graph Walking algorithm from [49] may improve optimization results.

Other techniques, which are partly already done, include procedure splitting, intraprocedural and interprocedural code positioning.

An interesting technique used in aiPop is procedure un-inlining – commonly repeated patterns in code are searched and a general procedure is made from them, resulting in smaller size.

87. This optimization may use the empty variable analysis to find out if a variable is used at a certain point.

88. An alias of a variable is another variable that contains the same value.

89. Such branches are actually quite often.

6.8.7 Improving cache performance

Data prefetch instructions could be generated before loops. This has to be done carefully, because it can sometimes also hurt the performance.

6.8.8 Inter-procedural analysis

The SSA form will be mostly done on a function basis. The results of the analysis from different functions can be merged for a particular purpose, such as tracking of value propagation across function calls. For registers, this is straightforward, when using values on stack, the stack pointer position differences must be taken into account.

6.8.9 Peephole optimizations

Some optimizations may be done using a small window ("peephole") inside which instructions are analysed and common patterns replaced.

Such optimizations could for example remove subsequent `PUSH`/`POP` pairs that may result from earlier stages (such as function inlining) or replace `PUSH` chains by `MOV` instructions and a single update of the stack pointer.[90]

90. This particular `PUSH`/`POP` optimization should not be done on processors with a stack engine.

7 Conclusion

The *IA-32 Binary Optimizer* framework presented in this thesis shows, that it is indeed possible to work with binary programs on Linux, even if they are stripped.

Some encouraging results have been obtained from programs, that were compiled and optimized for the particular processor on which they ran. The performance of such programs has sometimes been increased over 1.5%.

For programs packaged with the Linux distribution, such as the `rar` program, which were not compiled specifically for the target machine, the impact of optimization has been almost 5% on some processors. The `python` interpreter has been optimized even over 5% on the Pentium 4 processor.

With the SSA form, which should be soon available, even bigger gains may be expected, because more information about the code will be known. The Sun Binary Optimizer, which is a recent optimization system for Solaris on Sparc processors and uses similar optimizations targeted at cache usage, reports performance increases of up to 10%. It uses additional information about the code from the compiler. Even though it uses a different processor architecture, a similar improvement might be possible, considering the similarities in caching mechanisms. Other optimizations not related to cache usage will provide additional performance increase.

Anyone can develop new optimization plugins, which are relatively independent of the rest of the system, are easy to write and may perform almost any change to the code. Many suggestions for new optimizations have been proposed in *Chapter 6*.

The framework also offers many possibilities outside of optimization – it can be used to instrument almost arbitrary programs to gather profiling information. With little work, other information may be gathered from the program on the fly, security self-checks can be inserted into a third-party program.

Appendix A Program source code reference

The source code for the *IA-32 Binary Optimizer* is organized into several subsystems located in different directories as follows:[91]

- `common` – Common definitions, macros and simple utilities, like die(), basic data types (`vm_offset_t`, `u8_t`, `s8_t`, `u16_t`, `s16_t`, ...).
- `convertor` – Code for cloning a section and maintaining a connection between the original and the cloned sections. Also contains code for relocating and fixing the converted code.
- `decoder` – Instruction and basic block decoder and basic functions for manipulating code in basic blocks and sections.
- `decoder/parts-IA32` – x86 architecture-dependent code.
- `decoder/parts-AA64` – x64 architecture-dependent code.
- `instrumenting` – Support for instrumenting a program with counters and potentially other information. The actual definitions of the data structures used for instrumenting are in the `profiling` directory.
- `instrumenting/helpers-i386` – x86 architecture-dependent code.
- `instrumenting/helpers-AA64` – x64 architecture-dependent code.
- `optimizer` – Code analysis and optimization, handling of optimizer plugins.
- `optimizer/plugins` – The actual optimizer plugins.
- `profiling` – Definitions of counters and other profiling information.
- `sysdep` – Code dependent on the operating system and binary file format.
- `tests` – Self-tests of the framework. Not required by any other subsystem.
- `utils` – Universal data structures and more complex utility classes.

The `decoder` and `instrumenting` subsystems contain a (relatively small) part that is processor architecture dependent. Currently, only x86 code is supported, but parts of the framework for x64 are also written.

91. The description applies to the 1.0 version released with this Thesis.

Code that depends on binary file format or operating system is mostly placed into the `sysdep` directory, except small fixes in the convertor, which contain special cases, such as a list of "dangerous" symbols that should be left at their original locations.

Some of the most important source files, classes, functions and data structures will be described in the rest of the appendix.

A.1 Instructions

The `Instruction` class represents a single parsed x86 instruction. Its declaration and implementation are in `decoder/Instruction.[h|cpp]` files. This structure is filled from raw data every time information about instructions is needed.

```
struct Instruction {
    /* Default setting of the segment (such as operand or address size) */
    const SegmentSettings *settings;

    /* Pointer to a structure describing the instruction */
    InstructionInfo *info;

    /* Start address of the instruction */
    vm_offset_t address;

    /* Instruction opcode as in the binary code */
    int opcode;

    /* Total instruction length including all prefixes */
    byte total_length;
...
};
```

It additionally contains prefix information, ModR/M and SIB bytes, immediate data and displacement.

The `settings` field contains the default addressing mode of the code segment. The addressing mode can change the interpretation of the whole instruction, that is why it needs to be referenced by every `Instruction` object.

The `opcode` field is an internal representation of the 1 or 2-byte raw instruction code, without prefixes or ModR/M and SIB bytes. For instructions with a single-byte opcode, it is the opcode directly, for instructions with a two-byte opcode starting with the `F0H`, it is 0x100 plus the second byte of the opcode. Some instructions use some bits of ModR/M byte or a prefix as part of the opcode, but these do not influence the `opcode` value.

The `info` field points to a description of the instruction type in a `InstructionInfo` structure. It specifies a single variant of an instruction with description of instruction operands in the `op` array. The type of the operand is defined by on `Operand` class, which specifies a register or addressing mode and a generic size specification. The actual size of the operands depends on the model of the code segment and can only be determined using various other information, such as `SegmentSettings` and operand-size or address-size prefixes.

There are various methods in the `Instruction` class which can be used to analyse and modify the addressing mode, change operand sizes, displacement or immediate data. Other methods include queries and modifications of branch instructions or creating a new branch or a `NOP` instruction.

The `InstructionInfo` class is defined in `decoder/InstructionInfo.[h|cpp]` files and the definition looks like this:

```
struct InstructionInfo {
    /* Instruction type code (such as MOV or ADD) */
    const enum InsnCode code;

    /* Description of the operands */
    const struct Operand op[MAX_SPEC_OPERANDS];
...
};
```

The `code` field corresponds to the instruction mnemonic. It distinguishes between different instructions, but not when they differ only in operand size or type, such as register and memory operands. Every type of an instruction has its own `InstructionInfo` definition.

For example, all `MOV` instructions share the same `InsnCode`, but may have several `InstructionInfo` objects associated. On the other hand, all `MOV` instructions from memory to a full-size register share a common `InstructionInfo` object.

When additional information about an instruction is required, the `InstructionType` class may be used. It is defined in `decoder/InstructionType.[h|cpp]`.

It is a more detailed description of an instruction. It contains details, such as explicit and implicit operands or used and modified flags.

```
class InstructionType {
    /* Instruction type code (such as MOV or ADD) */
    enum InsnCode code;

    /* List of all source and destination operands */
    Vector<OperandRef> srcOps;
    Vector<OperandRef> dstOps;

    /* Mask of all used and modified flags */
    u32_t readsFlagMask;
    u32_t writesFlagMask;

    /* Other properties of the instruction */
    u32_t specialFlags;

    /* All variants of the instructions with the same code */
    Vector<InstructionVariant> variants;
...
};
```

There is one `InstructionType` object for one `InsnCode` code. It contains references to all `InstructionInfo` variants with the same `InsnCode` code. This can be used to modify an instruction to use different operand types, such as a register instead of memory or vice-versa.

A.2 BasicBlock class

The `BasicBlock` class represents a basic block – that is, a continuous stretch of instructions without a jump or with a jump only as the last instruction, in which the control flow can only start at the beginning.

The basic block definition used in the program does not even allow a `CALL` inside the block, so basic blocks are also split after `CALL` instructions. This restriction is added because when a block changes its address, all jump instructions that jump from or to the block need to be updated. If a basic block is defined this way, the only instruction that may use addresses into the code is the last one, so the update needs to be performed only on the last instruction in a block.

A `BasicBlock` object doesn't contain any parsed instructions, only raw data. If instructions are needed, they are parsed on the fly, because it is possibly cheaper than maintaining the list in a data structure – and it doesn't use as much memory.

The `BasicBlock` class contains many methods for modifying data of the block, splitting the block at the start, at the end or inside, changing its control flow and position among other blocks. There are also utilities that invert the condition of a conditional branch, change branch target or relax a branch.[92]

The `BasicBlock` class definition (`decoder/BasicBlock.[h|cpp]`) looks like this:

```
class BasicBlock {
    /* Start and end addresses of the block */
    vm_offset_t vm_start;
    vm_offset_t vm_end;

    /* Raw section data */
    const byte *data;

    /* Link to parent section */
    ProgramSection *section;

    /* Link to next block */
    BasicBlock *next;

    /* Link to a block that is a branch target */
    BasicBlock *nextBranch;
...
};
```

Every `BasicBlock` is assigned to a `ProgramSection` object contained in the `section` field. The `data` field points to raw data of the basic block. It uses memory allocated and managed by its containing section. When the basic block data changes, memory for the new data has to be allocated from the section.

The block's `ProgramSection` object takes care of all data modifications and gets notified when anything in the block changes. It also updates mappings from original addresses to new addresses.

92. Relaxing a branch is changing the size of the branch instruction to use the shortest possible encoding.

A.3 Function class

A `Function` class marks detected functions in the code. There is no need to detect functions in most plugins, so this information is optional. The optimizer plugins may use functions to improve optimization or to restrict the scope of changes to a function.

The source code is located in `decoder/Function.[h|cpp]` and looks like this:

```
class Function {
    String name;

    /* Link to parent section */
    ProgramSection *section;

    /* Start and end addresses of the function */
    vm_offset_t vm_start;
    vm_offset_t vm_end;

    /* Lists of calling functions and function called. */
    SimpleSet<Function *> callers;
    SimpleSet<Function *> called;
...
};
```

It contains the `name` of the function, information about whether it was possible to detect it from symbols or otherwise, its address range and a set of all callers and called functions.

A.4 ProgramCode and ProgramSection classes

The base class for the whole program code is the `ProgramCode` class. It contains some global information and a list of all sections. Each section[93] is represented by a `ProgramSection` object. Source code for both classes is in `decoder/`.

The `ProgramSection` provides a lot of methods for `BasicBlock` objects, such as allocation of memory for block data, creating new blocks and maintaining their original memory mapping.

A `BasicBlock` notifies its section of all changes of its contents, such as adding, removing, moving or changing a part of code. The section notifies all registered listeners about the change. One of the listeners updates symbols and relocations and moves or removes them according to changes in the code they are associated with.

A listener called `CodeTracker` tracks all modifications to the code to maintain a mapping from original VMA[94] before code cloning, instrumenting or optimization to current VMA addresses and vice-versa. It gets notified of every change, so that the mapping is always up-to-date.

The base class of all content change listeners is `BasicBlockManager`. It contains virtual methods for notification of changes in the code. The concept of listeners was introduced because the code in the `decoder` is written to be universal. This way, tracking of content movement can be added as an optional part.

93. This usually means an ELF section, but the `decoder` is independent of the object file format.

94. VMA stands for Virtual Memory Address.

There is also a list of local symbols and relocations. They are used to locally mark and relocate content, but they are never used outside of the `decoder`.

A `ProgramSection` object may be in one of two states. It is either without any blocks parsed, containing only raw section data, or all data are stored within parsed blocks. For sections that are read from the original file and not processed by the instrumenting or optimization, only the raw data is used. In this case, `mem_size` stores the size of the section in memory. For sections such as `.bss`, it can differ from the size in the file, so this information must be transferred to the final program.

If there are any blocks in a `ProgramSection` object, raw data from the `data` field are ignored and only data from the blocks are used.

All data allocation for basic blocks is done using the `SectionDataBuffer` object from the `allocator` field, raw data are managed by the `DataBuffer` object `data`.

The definition of `ProgramSection` (`decoder/ProgramSection.h`) looks like this:

```
class ProgramSection {
    String section_name;

    /* Link to parent object */
    ProgramCode *code;

    /* Default setting of the segment (such as operand or address size) */
    SegmentSettings settings;

    /* Start address of the section */
    vm_offset_t vm_start;

    /* Buffer for all raw data (only used without blocks) */
    DataBuffer data;

    /* Section size in memory (only used without blocks) */
    u32_t mem_size;

    /* First basic block (NULL if there are no blocks) */
    BasicBlock *first;

    /* Allocator for new basic block data */
    SectionDataBuffer allocator;

    /* Allocator for basic blocks (they can be polymorphic) */
    BasicBlockFactory *factory;

    /* Map of all basic blocks */
    AVLTreeMap<vm_offset_t, BasicBlock *> blocks;

    /* Symbols, relocations and functions */
    HashMap<BasicBlock, Symbol *> localSymbols;
    HashMap<BasicBlock, Relocation *> localRelocations;
    AVLTreeMap<vm_offset_t, Function *> functions;
...
};
```

The `ControlFlowGraph` class holds a control flow graph of a section.

A.5 ProgramConvertor and SectionConvertor classes

The code of the original file is converted to the destination file in both instrumenting and optimization. The subsystem responsible for the conversion and management of the connection between the original and new code is the *convertor*.

It uses the BFD library – unlike the `decoder`, which only takes care of the code with no relation to the original program files. The `convertor` is responsible for loading of the binary file, handling all chores associated with its modification, and finally writing the new program to a file.

The base class, representing the current state of the converted program and its links to the original and new file, is the `ProgramConvertor`. It contains a list of BFD symbols and dynamic symbols, BFD file handle of the original and destination file and other information used in the BFD library. Additionally, it maintains a list of all sections as `SectionConvertor` objects.

The `ProgramConvertor` class (`convertor/ProgramConvertor.h`) looks like:

```
class ProgramConvertor {
    /* Link to the corresponding object in decoder */
    ProgramCode code;

    /* All sections (including non-text) */
    SingleList<SectionConvertor *> sections;

    /* List of all symbols and dynamic symbols */
    SingleList<SymbolInfo *> symbols;
    SingleList<SymbolInfo *> dSymbols;

    /* BFD handle of the source and destination program */
    bfd *source;
    bfd *dest;

    /* BFD global program flags */
    flagword flags;

    /* Entry point of the program */
    bfd_vma start_addr;
...
};
```

It corresponds to the `ProgramCode` class from `decoder` which also represents a whole program, but includes additional information needed for conversion.

A `SectionConvertor` object is similar to a `ProgramSection` object from `decoder`. It contains a reference to a `ProgramSection` object, and some BFD-specific data, such as flags, relocations and BFD section handles. It has also many proxy methods to `ProgramSection`, original to current address translation routines, and functions for reading, modification and writing of section data. Other functions include searching for blocks and functions, creating, removing and moving blocks around.

The `SectionConvertor` class (`convertor/SectionConvertor.h`) looks like this:

```
class SectionConvertor {
    /* Reference to a ProgramConvertor object representing the whole file */
    ProgramConvertor *convertor;

    /* Reference to the ProgramSection object (contains the blocks) */
    ProgramSection section;

    /* Manages BasicBlocks - gets notification of their changes */
    CodeTracker *tracker;

    /* Original VMA of the section - as set by setOriginalVMA() */
    vm_offset_t orig_vma;

    /* BFD handle of the source and destination sections */
    asection *bfd_section;
    asection *dest_bfd_section;

    /* BFD flags of the section */
    flagword flags;

    /* List of all BFD relocations */
    SingleList<RelocationInfo *> relocations;
...
};
```

Additionally, there is a `CodeTracker` class which is registered at a `ProgramSection` object, receives notifications about changes and maintains a correspondence between source and destination addresses as the code changes.

Relocations, which correspond to BFD relocations, are stored in `RelocationInfo` objects.

A.6 Code conversion

Both instrumenting and optimization involve copying code from a source file to a destination file, with some modifications. In the first case, an initialization routine in the C language and counter code are also included in the result.

The `ConversionHelper` is responsible for analysing the program, copying the code from the original code section into the new section and parsing it.

It also handles all fixes to the conversion process, such as:

- inserting trampoline jumps from original code to new code
- fixing ELF program headers (BFD sometimes produces a defective file)
- producing debug line information for the new code
- creating exception handler frames for the new code[95]

95. This feature is being worked on and is not yet finished.

The `ConversionHelper` class (`convertor/ConversionHelper.h`) looks like this:

```
class ConversionHelper {
    /* A ConversionHelper is tied to a single ProgramConvertor */
    ProgramConvertor &convertor;

    /* Bitmap of machine instruction start addresses */
    VMABitmap instructionStarts;

    /* Information about possible jump destinations in the original section */
    HashMap<vm_offset_t, TargetInfo *> targetsInfo;

    /* Map from VMA addresses to symbolInfo structures */
    HashMap<vm_offset_t, SymbolInfo *> symbolVMAMap;

    /* VMA addresses where basic blocks should be forced to end */
    VMABitmap blockBreakLocations;

    /* VMA addresses where trampolines should be added */
    VMABitmap trampolineLocations;

    /* VMA address of the start and end of the original section */
    vm_offset_t code_start;
    vm_offset_t code_end;
...
};
```

Name of the new section and alignment of the its start and end is configurable in the `ConversionHelper` object.

Another class, `SectionBlockManager` (from `convertor/SectionBlockManager.h`), takes care of reordering blocks. It is able to add or remove jumps at the end of the blocks or invert branch conditions to optimize the reorganized code.

The code to be included as counters or the initialization routine is stored in an object file as different sections, so that different implementations might be provided without altering code of the rest of the framework. For reading the file, `SimpleObjectFile` and `SimpleObjectSection` classes are used. They are defined in `convertor/SimpleObjectFile.h` and `convertor/SimpleObjectSection.h`.

A.7 Code tracking

There is a mechanism for tracking code movement. This is necessary to support features such as trampolines, producing debug line and exception frame information for the new code.

The `CodeTracker` class (defined in `convertor/CodeTracker.h`), which inherits from `SectionBlockManager`, implements this tracking.

It maintains a map from `BasicBlock` objects to a `ContentTraces` object associated with the block. One `ContentTraces` object manages address tracking information for one basic block. It is a list of `ContentTrace` objects, which hold data about one span of the block: its offset, length and original VMA address. This data is used for translating addresses from new to original addresses.

There is also a map from original addresses to `BasicBlock` objects, which enables the translation in opposite direction. A special data structure called `AVLSpanTreeMap` is used for this translation. It is able to search for keys which are VMA slices.

A `CodeTracker` object is registered with a `ProgramSection` object to be notified about all changes to the code, so it can update the database of address mappings on the fly.

A generic interface to a VMA translator is also provided in the `VMATranslator` class. It uses a `CodeTracker` object to provide the translations.

A.8 Instrumentation and profiling

Instrumenting uses simple 8-byte `BBCounter` data structures as counters. Every counter also has a `BBCounterInfo` structure describing it.

The `BBCounter` structure (`common/SectionHeaders.h`) holds the actual counter:

```
struct BBCounter {
    u32_t counter_lo; /* Low part of the counter - inverted */
    u32_t counter_hi; /* High part of the counter */
};
```

The `BBCounterInfo` structure (`common/SectionHeaders.h`) has information about a block and its counters:

```
struct BBCounterInfo {
    vm_offset_t orig_vma; /* Original VMA address of the block with counter */
    vm_offset_t dest_vma; /* Destination VMA address of the block with counter */
    u32_t prologue_offset; /* Offset of the prologue counter in counters */
    u32_t epilogue_offset; /* Offset of the epilogue counter in counters */
};
```

The `CounterMap` class (in `profiling/CounterMap.h`) looks like this:

```
class CounterMap {
    /* Map from VM addresses to counters info structures */
    AVLTreeMap<vm_offset_t, BBCounterInfo *> countersMap;

    /* MMaped counter values file. Holds the BBCounter values of the counters. */
    IA32BinOptCountersHeader *bssHeader;

    /* Info header of the counter info section with basic info about counters. */
    IA32BinOptInfoHeader infoHeader;

    /* Contents of the counter info section. Holds BBCounterInfo objects */
    byte *contents;

    /* Are the VMA addresses original or destination addresses? */
    bool usingOriginalFile;
...
};
```

In `countersMap`, it contains a map from original or destination addresses[96] to `BBCounterInfo` counter information objects. From them, it is possible to get the actual counter value stored in a `BBCounter` object.

96. This depends on value of the `usingOriginalFile` flag.

The main instrumentation code is in `instrumenting/prepare_counting.cpp`.

A.9 Optimizer support structures

Optimizer plugins typically need more information about the code than the standard `ProgramSection`, `BasicBlock` and `Instruction` objects can offer. They are provided with additional data in a `OptimizeContext` object.

The `OptimizeContext` class (`optimizer/OptimizeContext.h`) looks like this:

```
class OptimizeContext {
    /* The ProgramConvertor object representing the optimized program */
    ProgramConvertor *programConvertor;

    /* The section selected for optimization */
    SectionConvertor *selectedSection;

    /* List of all basic blocks */
    DListT<BasicBlockInfo> blocks;

    /* Map of counters for all blocks */
    const CounterMap *counterMap;

    /* Additional information for optimization, such as a CFG or SSA form */
    OptimizeFramework *optimizeFramework;
...
};
```

It contains a reference to the code section,[97] a separate double-linked list of all `BasicBlock` objects in the `blocks` field for easy reorganization, reference to a counter map to query about pass counts of basic blocks and an optional reference `optimizeFramework` to an `OptimizeFramework` object, which contains additional information.

The actual `BasicBlock` objects are wrapped in `BasicBlockInfo` structure, which provides additional information for the basic block, such as data about a counter or a more complete control flow information.[98]

The `BasicBlockInfo` objects are also used to reorganize the blocks. While the `BasicBlock` class has the `next` field to chain blocks, it cannot be easily used to shuffle the blocks, because it doesn't only determine the position of the block, but also its control flow.

After the `BasicBlockInfo` objects are shuffled, they can be passed for processing to a `BasicBlockManager` object which updates their position and control flow data.

Another information stored there is a reference to a counter information for the block. The counter value cannot be directly obtained from it, however. It needs a `CounterMap` object, which contains the values.

97. Every `OptimizeContext` works on a single section or a part of it.

98. The `BasicBlock` object is a more low-level object and only contains a reference to the next block in normal control flow and a branch block for direct jumps.

The `BasicBlockInfo` class (`optimizer/BasicBlockInfo.h`) looks like this:

```
class BasicBlockInfo: public DNode {
    /* The basic block represented by this object */
    BasicBlock *block;

    /* Reference to a counter info for this block */
    const BBCounterInfo *counter;

    /* Next basic block - next in in control flow, not necessarily in code */
    BasicBlockInfo *nextBlockInfo;

    /* Pointers to all branch blocks */
    Vector<BasicBlockInfo *> branchBlocksInfo;
...
};
```

The `OptimizeFramework` is constructed for a section or possibly for a part of it, such as a function. It contains a `ControlFlowGraph` object, which may be considered redundant, as `BasicBlockInfo` objects already contain this information, but this allows using generic graph algorithms on the control flow graph of the code.

The `OptimizeFramework` class (`optimizer/OptimizeFramework.h`) looks like this:

```
class OptimizeFramework {
    /* The section for which the object is constructed */
    SectionConvertor *section;

    /* Working set of the blocks, usually a function */
    BasicBlockSet *workSet;

    /* Entry point of the working set, eg. of a function */
    BasicBlockWI *rootBB;

    /* Array of all blocks in the working set */
    Array<BasicBlockWI *> blocks;

    /* Control-flow graph of the section */
    ControlFlowGraph CFG;

    /* The SSA form of the code */
    SSAForm ssa;
...
};
```

The `workSet` field is a reference to a `BasicBlockSet` object, which determines the set of blocks that should be processed.

The basic block objects used in the optimizer are not instances of the `BasicBlock` class, but instead of its subclass `BasicBlockWI`. A `BasicBlockWI` object contains additional control flow information, such as a reference to the previous block or multiple branch blocks, which may be used for indirect jumps. It also contains invariants and free locations at the start and at the end of the block, its immediate dominator and post-dominator, dominance frontier and other information used while building the SSA form.

The `BasicBlockWI` class (`optimizer/BasicBlockWI.h`) looks like this:

```
class BasicBlockWI: public BasicBlock {
    /* List of all instructions */
    DoubleList<InstructionInstanceWI *> instructions;

    /* Pointer to the previous block */
    BasicBlockWI *prev;

    /* Invariants at the start/end of the block */
    InvariantsInfo inputInvariants;
    InvariantsInfo outputInvariants;

    /* Temporary invariants that will need to be merged with other branches */
    InvariantsInfo inputInvariantsMax;
    InvariantsInfo outputInvariantsMax;

    /* Free locations at the start/end of the block */
    LocationsInfo inputLocations;
    LocationsInfo outputLocations;

    /* Block number for the DFS/BFS search (search number) */
    int blockNumber;

    /* Immediate dominator */
    BasicBlockWI *idom;

    /* Immediate post-dominator */
    BasicBlockWI *ipdom;

    /* This block's dominance frontier */
    SingleList<BasicBlockWI *> dominanceFrontier;

    /* List of variables that can get into the block. */
    SimpleSet<MergeVariable> mergeVariables;

    /* Flags used in SSA construction */
    int flags;
...
};
```

In the `instructions` field, there is also a list of `InstructionInstanceWI` objects representing parsed instructions and additionally containing invariants and empty locations in `LocationsInfo` and `InvariantsInfo` objects.

In future, this class might be removed and replaced by a mapping between basic blocks and the additional information it now contains. This would have the advantage that when the data are not needed any more, they can be easily discarded.

The `SSAForm` class should contain the SSA form of the code, but it is not finished yet.

The `ProcessorInfo` (defined in `optimizer/ProcessorsInfo.h`) and `SystemInfo` (defined in `optimizer/SystemInfo.h`) classes provide information about the CPU and operating system which runs the code.

A.10 Invariants

Some optimizer plugins need additional information about the properties of the code at certain points. The `LocationsInfo` class holds information about free and used places on stack and in registers and `InvariantsInfo` holds all known invariants (or properties) about the code. It can store information, such as what is the value stored in a register, what value is for sure not stored in a register or if a value is lower than another value, lets say, on stack.

The `LocationsInfo` class (`optimizer/LocationInfo.h`) looks like this:

```
struct LocationsInfo {
    /* Set of unused registers and their parts */
    SimpleSet<RegID> unusedRegisters;

    /* Set of unused stack spans */
    SpanSet<s16_t> unusedStack;

    /* Bitmask of unused flags */
    u16_t unusedFlags;
...
};
```

The `unusedRegisters` field is able to track arbitrary parts of a register, the `unusedStack` field contains spans of bytes on stack that are not used. Both data structures allow fast searching if a particular location or part of it is in use or free.

Invariants store information which can formally be deduced from previous code. If, for example, there is a comparison of the EAX register with a number 5 and then a `JZ` branch, an invariant that EAX is 5 can be introduced in one branch and that it cannot be 5 can be put into the other branch.

In case some code has multiple control flow paths that can lead to it, an intersection operation of invariants from all paths must be performed.

The `InvariantsInfo` class (`optimizer/InvariantsInfo.h`) looks like this:

```
class InvariantsInfo {
    /** A set of invariants for registers and stack */
    SingleList<Invariant> invariants;

    /** Bitmask of flags that are known to be set/clear */
    u16_t flagsSet;
    u16_t flagsClear;

    /** Current stack and FPU stack position relative to a base */
    s16_t stackPos;
    s16_t FPUStackPos;
...
};
```

One `Invariant` (`optimizer/Invariant.h`) has 2 operands, both can be a register or a memory location, and the second one can also be a constant. The invariant then stores a type of relation between the two values. The relation may be an equality, inequality, other comparison, or an information on bits which are set or clear.

The registers or stack memory locations used in the invariants are represented by a `VariableID` class (`optimizer/VariableID.h`). It can represent a register, part of a register or a stack memory area.[99]

While the location tracking is already implemented, the invariants and SSA form construction are both work in progress and are not in a usable state yet.

A.11 Optimizer plugins

All optimizations are performed by optimizer plugins. They are all descendants of the base class `OptimizePlugin` (`optimizer/OptimizePlugin.cpp`). It contains a configuration of the plugin and a virtual method *optimize()*, which does the actual work. The plugins, such as `CacheUnaliasPlugin`, `AthlonBTBPlugin` or `BranchAlignPlugin`, are located in the `optimizer/plugins` directory.

All plugins are registered at an `OptimizePluginRegistry` object (the definition is in `optimizer/OptimizePluginRegistry.h`). This object manages all plugins and is responsible for reading their configuration from a config file.

The main code of the optimizer is in `optimizer/optimize.cpp`.

A.12 System-dependent parts

The code which depends on the operating system or object file format is located in the `sysdep` directory.

The base of the system dependent part is the `ObjectFile_utils` class. Currently, only one implementation is provided, for the ELF file format and the Linux operating system – `Elf32_utils`. It contains code that fixes an ELF file after it is written by the BFD library. This is necessary because the BFD library, when used in a different way than normally used, sometimes produces executable files that don't work properly. Sometimes, for example, `LOAD` segments are not produced for some loadable sections or are aligned in a wrong way.

The is also a `DwarfUtils` class in the `sysdep/dwarf` directory. It adjusts debug information in the DWARF2 format to cover the new code, either instrumented or optimized.

A.13 Containers and other universal data structures

There is a large number of supporting data structures and other common code in the `utils` directory. Some of the classes have a similar function as classes in the STL C++ library. A separate version was created to improve debugging possibilities and add some functionality that was not present in the STL classes.

For some data structures, different implementations were considered to measure performance and select the best data structure for the task, some might be better described as pet projects.

99. It can represent up to 127 bytes in the range [stack_base - 32768, stack_base + 32767].

Several universal containers are defined. `Array` is a simple array with bounds checking wrapped in a class. `Vector` is similar to `Array`, but allows pushing new elements at the end or inside the array and provides automatic resizing. `Heap`, `Stack` and `Queue` are simple array-based implementations of a heap, stack and queue.

There are several implementations of linked lists. They fall into several categories, some are single-linked, some are double-linked. Some use a fixed size of the nodes, some occupy less memory at the expense of restricting adding new nodes only at the beginning (the classes with the "*Simple*" prefix), some allocate data for the node, some use the actual data as a node and require it to inherit from a base class (the classes *SList**, *DList**). The list classes are: `SList`, `DList`, `SingleList`, `DoubleList`, `SimpleSingleList`, `SimpleSList`.

There are several versions of a universal set, each with different capabilities and efficiency for particular operations: `AATreeSet` implemented as an AA tree, `HashSet`, which uses a hash table, `RBTreeSet` implemented as a red-black tree, `SimpleSet` stored in a sorted array.

There are also several map implementations: `AVLTreeMap` is using an AVL tree, `HashMap`, `RBTreeMap`, `SimpleMap`, are based on corresponding sets, `SplayTreeMap` is using splay trees. `StringMap` and `StringIntMap` use strings as keys.

Several sets and maps are implemented which use a span of values as a key and allow interval searching: `SpanSet` (using `SimpleSet`) and `AVLSpanTreeMap` (using `AVLTreeMap`).

`SimpleObjectSet` is for storing objects implementing a particular interface. The `EvalLinkSet` is an implementation of an eval-link set from [39], used for dominator analysis. `HashMemberStack` and `HashMemberQueue` are hybrid data structures, which combine a hash set with a stack or queue.

For control flow graph and other graphs, there is a `Graph` class for bidirectional graphs and `SingleGraph` for directed graphs.

Bitmaps can use the simple `Bitmap` class or `VMABitmap` for VMA address bitmap, `Flags` when multiple bits per record (flags) are needed instead of a 1-bit bitmap, `VMAFlags` for VMA address flags. `SimpleFlags` is for flags that fit into an integer.

There are other special classes: `ConfigFileParser` which is used for parsing optimizer config files, `String` for storing dynamically changing character strings, `PrintBuffer` for string formatting, `IOBuffer` for reading line-buffered data from files, `DataBuffer` for universal data storage used in `ProgramSection`.

Other utilities include `ArrayUtils` for sorting arrays, `EndianUtils` for conversion of data between endianities, `NumericUtils` for working with rounding and prime numbers and `FileUtils` containing methods dealing with file paths.

Appendix B Usage of the tools

B.1 Building and installation of the tools

If you want to build the *IA-32 Binary Optimizer* from sources, download the source file package `ia32_binopt-<version>.tar.bz2` and possibly additional libraries `libs.tar.bz2` and then run these commands:

```
tar xjf ia32_binopt-<version>.tar.bz2
cd ia32_binary_optimizer
tar xjf ../libs.tar.bz2
make -f Makefile.cvs
mkdir out; cd out
../configure --enable-static-bfd --enable-static-iberty
make
```

If you have a BFD version at least 2.16.91, you can skip unpacking of the additional libraries and omit the `--enable-static-bfd` and `--enable-static-iberty` flags to the configure script. The supported version of dietlibc is 0.30.

Alternatively, you can install the provided binary RPM package.

B.2 Configuration file

The optimization is directed by a configuration file. It consists of a list of configuration options organized into sections. A configuration option consists of a name and a value, separated by a colon ':'. The list of sections and options in each section, as well as types of values, are fixed and unknown values are reported as warnings.

There is one global configuration section called `PluginsConfig`. The names of other sections are the names of the corresponding optimizer plugins.

Common options supported by most optimizer plugins are `enabled` and `debug`. Both have boolean values and control whether the plugin is enabled and whether some verbose debugging messages should be printed. The possible configurations for each optimizer plugins are presented below.

If the configuration file is not specified on the command-line, a `ia32binopt.conf` file in the directory with the instrumented file is used. If there is no such file, the directory with the `ia32bopt_optimize` program is tried next. As the last resort, the file `${HOME}/.ia32binopt.conf` is used.

B.2.1 CacheUnalias plugin

The CacheUnalias plugin compacts hot code in hot-spots by moving or copying together blocks that frequently pass control among each other, optimizing code cache usage.

The percentage of the maximal basic block passes count, which should be considered a hot-spot (and optimized), is determined using the `percentTreshold` option. If `mergeHotSpots` option is enabled, different hot-spots are placed together if they use each other frequently.

Normally, when compacting hot code, basic blocks are moved, not copied, from their original location. If `enablePartialInlining` is true, then some blocks may be either moved or copied from another part. The `partialInlineTreshold` controls how many percents of passes must come into block B from block A, so that B is moved next to A. If there are less passes coming from from A, then B is only copied next to A and also stays at its original location.

B.2.2 BranchAlign plugin

The BranchAlign plugin aligns important targets of jumps (branches) to a multiple of `alignSize` (usually 16), which improves performance on some processors.

The percentage of the maximal basic block passes count, which should be considered a hot-spot and optimized, is determined using the `globalPercentTreshold` option.

To select branch targets that should be aligned, the `jumpAlignTreshold` option can be used. There must be at least `jumpAlignTreshold` times more jumps to the address than simple passes through from the previous basic block in order to align the block. The alignment is done to a multiple of `alignSize`.

The padding can be up to `alignSize` - 1 long, and it is filled with `NOP` instructions.

If the padding is long, it may slow down normal passage to the block. A jump can be inserted at the beginning of the padding to the next block if `jumpOverPadding` is enabled. The minimal padding size where a jump should be inserted is controlled by `jumpOverPaddingLength`.

Jumps generated over padding seldom improve performance and therefore should not be used too often. If the padding hurts performance of the normal control flow, maybe the `jumpAlignTreshold` option is set too low and there should not be any padding at that location in the first place.

B.2.3 AthlonBTB plugin

The AthlonBTB plugin increases the size of some `RET` instructions to solve problems with Athlon BTB cache.

There is only one configurable option, `globalPercentTreshold`, which controls which blocks should be affected by the optimization.

B.2.4 HotColdSeparate plugin

The HotColdSeparate plugin separates *hot* code from *cold* code. It is similar to the CacheUnalias plugin, but it is simpler, because it doesn't work at a basic block level, but moves whole functions.

There are no configuration options, except for the standard `enabled` and `debug`. It doesn't make sense to separate hot and cold code only in a hot-spot.

B.2.5 A sample configuration file

A sample configuration file may look like this:

```
[CacheUnaliasPlugin]
enabled: yes
debug: yes
percentTreshold: 15.0
mergeHotSpots: true
enablePartialInlining: true
partialInlineTreshold: 90.0

[BranchAlignPlugin]
enabled: no
debug: yes
globalPercentTreshold: 80
jumpAlignTreshold: 60
jumpOverPadding: true
jumpOverPaddingLength: 5
alignSize: 16

[AthlonBTBPlugin]
enabled: no
debug: yes
globalPercentTreshold: 80

[HotColdSeparatePlugin]
enabled: no
debug: yes
```

B.3 Command-line options

All tools support the `--help` option which will write a summary of possible options and, in the case of ia32bopt_optimize, also default values.

B.3.1 ia32bopt_prepare

The `ia32bopt_prepare` program is used for instrumenting a file intended to be optimized. The command-line options are summarized in the table below:

File names:	
`--input-file` <*filename*>	Input file name to be instrumented
`-f` <*filename*>	Same as `--input-file`
`--output-file` <*filename*>	Output file name (instrumented program)
`-o` <*filename*>	Same as `--output-file`
`--code-utils-file` <*filename*>	Helper file with architecture-specific code
`-c` <*filename*>	Same as `--code-utils-file`
Code handling options:	
`--program-headers-fix=`<y/n>	Fix ELF program header table
`--page-align-section=`<y/n>	Align new code section to page boundary
`--page-align-section-end=`<y/n>	Align new code section end to page boundary
`--keep-empty-blocks=`<y/n>	Keep or delete empty blocks (filled by NOPs)
`--breaks-on-syms=`<y/n>	Split basic blocks on symbol locations
`--insert-trampolines=`<y/n>	Insert trampolines from old to new code
`--trampolines-on-syms=`<y/n>	Insert trampolines on symbol locations
`--trampoline-mode=`<*mode*>	Select trampoline placing algorithm
`--avoid-trampoline=`<*address*>	Avoid inserting a trampoline on *address*
`--allow-exceptions=`<y/n>	Force instrumenting a file using C++ exceptions
`--select-abi`	Select ABI – calling conventions
Other options:	
`--dump-section-info`	Print information about sections
`--print-targets`	Print available BFD targets
`--dump-xtable=`<*filename*>	Dump VM address translation table to file
`--att-syntax`	Use AT&T syntax for disassembly
`--help`, `-h`	Print a help screen

Table 3. Command-line options for ia32bopt_prepare

Trampoline placing algorith can be `immediate`, `delayed` or `anywhere`. ABI can be `normal`, `regparm` or `unknown`.

In most cases, standard options should be sufficient and only the input file has to be specified. If some problems with trampolines are reported, another trampoline mode may be selected. For the 2.4.x version of Linux kernel, it may be necessary to activate the option `--page-align-section-end`, because the kernel doesn't load code segments properly if they are not aligned to a page boundary.

If the output filename is not specified, a suffix "`.instrumented`" is appended to the original name.

B.3.2 ia32bopt_optimize

The `ia32bopt_optimize` program takes the instrumented file and optimizes it.

Not all options can be specified on the command-line. Most options dealing with optimization can only be stated in the config file which is selected using the option `--config-file`.

File names:	
`--input-file` <*filename*>	Input file name to be optimized
`-i` <*filename*>	Same as `--input-file`
`--output-file` <*filename*>	Output file name (optimized program)
`-o` <*filename*>	Same as `--output-file`
`--origial-file` <*filename*>	Original file name (usually automatically determined)
`-r` <*filename*>	Same as `--original-file`
`--code-utils-file` <*filename*>	Helper file with architecture-specific code
Code handling options:	
`--program-headers-fix=<y/n>`	Fix ELF program header table
`--page-align-section=<y/n>`	Align new code section to page boundary
`--page-align-section-end=<y/n>`	Align new code section end to page boundary
`--breaks-on-syms=<y/n>`	Split basic block on symbol locations
`--insert-trampolines=<y/n>`	Insert trampolines from old to new code
`--trampolines-on-syms=<y/n>`	Insert trampolines on symbol locations
`--trampolines-on-counters=<y/n>`	Insert trampolines at all places with counters
`--trampoline-mode=`<*mode*>	Select trampoline placing algorithm
`--avoid-trampoline=`<*address*>	Avoid inserting a trampoline on *address*
Optimization options:	
`--no-optimization`	Disable all optimizations
`--config-file=`<*filename*>	Configuration file for optimizer plugins
`--keep-empty-blocks=<y/n>`	Keep or delete empty blocks (filled by NOPs)
`--condense-blocks=<y/n>`	Put blocks as close together as possible
`--align-hot-blocks=<y/n>`	A switch to enable/disable the BranchAlign plugin
`--optimize-athlon-btb=<y/n>`	A switch to enable/disable the AthlonBTB plugin
`--select-abi`	Select ABI – calling conventions
Other options:	
`--disassemble-input`	Disassemble the code before optimization
`--disassemble-output`	Disassemble the code after optimization
`--disassemble-both`	Disassemble the code before and after
`--dump-block-map=`<*filename*>	Dump map of input and output basic blocks
`--dump-block-checks=`<*filename*>	Dump a map of blocks with origianl VMAs for checking
`--dump-xtable=`<*filename*>	Dump VM address translation table
`--att-syntax`	Use AT&T syntax for disassembly
`--help`, `-h`	Print a help screen

Table 4. Command-line options for ia32bopt_optimize

Options with the same name as for `ia32bopt_prepare` have the same meaning. The original file name may be specified, otherwise it is obtained by removing the "`.instrumented`" suffix from the instrumented file.

In most cases, standard options should be sufficient and only the input file has to be specified. If some problems with trampolines are reported, another trampoline mode may be selected. For the 2.4.x version of Linux kernel, it may be necessary to activate the option `--page-align-section-end`, because the kernel doesn't load code segments properly if they are not aligned to a page boundary.

If the output filename is not specified, the "`.instrumented`" suffix from the input file is replaced by "`.optimized`".

B.3.3 ia32bopt_analyse

The `ia32bopt_analyse` is a tool that visualizes the information obtained from profiling with the disassembly of the program. The input file is an instrumented program specified using the `--instrumented-file` or `-f`, optionally the original file can be entered with `--original-file` of `-o`. The AT&T syntax for disassembly can be requested by the `--att-syntax` option.

B.3.4 ia32bopt_disassemble

The `ia32bopt_disassemble` is a tool that disassembles a binary file, providing additional information. The input file is any binary executable program specified using the `--file` or `-f`. The utility can show recognized basic blocks and their control flow with the `--show-basic-blocks` option. Empty basic blocks can be suppressed by the `--remove-empty` option. Similarly to the other tools, AT&T assembler syntax for disassembly is selected by the `--att-syntax` option.

B.3.5 ia32bopt_cpuinfo

This program writes information about the CPU it runs on. There are no command-line options available.

Appendix C Example session

C.1 Preparation

In this appendix, a typical session with the optimizer will be described. We will be optimizing the Links browser.[100]

We start by compiling the *IA-32 Binary Optimizer* framework from sources or by installing a binary package – as described in section B.1.

If compilation fails because of dependency problems, running `make` again may help. To avoid incompatibilities between different versions of libraries, local copies may be used by specifying one or more of:

`--enable-static-bfd`	use the local copy of the `BFD` library
`--enable-static-iberty`	use the local copy of the `iberty` library
`--enable-static-dietlibc`	use the local copy of the `dietlibc` library

to the `configure` script. If you use any of these options, please make sure that you downloaded the *libs* package and unpacked it into the `libs` directory in the root of the source tree before running `make`.

Note that every time after running the `configure` script (from the `out` directory), the `make` command must be executed again (from the same directory).

We then compile the Links program and install it into `/usr/local/bin/links`. Now, we want to get the instrumented version, which will provide some profiling information.

C.2 Instrumenting

To instrument the program, we execute:[101]

```
ia32bopt_prepare /usr/local/bin/links
```

A new program, called `/usr/local/bin/links.instrumented` is created. When executed, it will create profiling information. By default, this data would go to `/tmp`, but we may want to use `~/.counters` instead, so we define the `IA32BINOPT_BASE` variable.

```
IA32BINOPT_BASE=~/.counters; export IA32BINOPT_BASE
```

When instrumenting a program which uses exceptions, the `--allow-exceptions=y` option to `ia32bopt_prepare` may be necessary. Note that the instrumented or optimized program may crash on the first thrown exception. This option should therefore only be used for programs that use exceptions solely for unrecoverable error conditions.

100. The homepage of the Links browser is at http://links.sourceforge.net/.

101. For this to work, `ia32bopt_prepare` must be on the path. Otherwise, the path to it must be put in front of the command.

As input for the Links browser, we prepare a test HTML page containing many different features we want to focus the optimization on. In our example, the test page will be placed in a file called `links-test.html`. The command which will produce profiling data is:[102]

```
links.instrumented -dump links-test.html > links-test.txt
```

We redirected the normal output of the progarm to see any error messages. We then check the error output of the program. If the instrumented binary did not start, there may be a problem in the instrumenting process.

If we use an older Linux kernel, we can add the `--page-align-section-end=y` option to the invocation of `ia32bopt_prepare`. This is normally the default option on Linux 2.4.x kernels, but the operating system version may be incorrectly detected.

If the instrumented program still doesn't run, other options to `ia32bopt_prepare` may be specified, such as changing the `--trampoline-mode` mode.

If everything worked well, a counter file should be placed under `~/.counters`.

We may run the `links.instrumented` program again with different data. If we want to optimize for a mix of usage patterns, we may create another test page in the `links-test2.html` file and update profiling data:

```
links.instrumented -dump links-test2.html > links-test2.txt
```

C.3 Analysis

To see profiling information gathered, along with some additional annotations, you can run:

```
ia32binopt_analyse -f /usr/local/bin/links.instrumented
```

A disassembly of the code divided into basic blocks and annotated with additional information is displayed. For every block, control flow data and counter value are displayed.

C.4 Optimization

We may now proceed to optimization. We copy the example configuration file `ia32binopt.conf` to the directory with the instrumented file and we run:

```
ia32bopt_optimize /usr/local/bin/links.instrumented
```

The optimized file will be placed in `/usr/local/bin/links.optimized`. To display some analysis data for optimization, the `--disassemble-both` command-line option may be provided. It will output similar data as the `ia32binopt_analyse` program, but additionally a dominator tree[103] and results of stack pointer analysis and empty locations analysis.

102. For this to work, `/usr/local/bin` must be on the path. Otherwise, it must be put in front of the command.

103. The dominator tree is only displayed, if `DEBUG_DOMINATORS` is set to 1 in `SSAForm.cpp`.

We then try to run it and measure its performance with the `time` command:

```
time /usr/local/bin/links.optimized -dump links-test.html >/dev/null
```

Then, we make some changes to the config file `/usr/local/bin/ia32binopt.conf` used in optimization and run `ia32bopt_optimize` again. We observe the impact of the changes on execution time and possibly repeat adjusting optimization options until we are satisfied with the result.

The first step in tuning the configuration file could be enabling or disabling certain plugins altogether and observing the impact on performance. Other common options that heavily influence optimization and can be tuned are `percentTreshold` and `enablePartialInlining` for the `CacheUnaliasPlugin` plugin.

Bibliography

[1] Matthew C. Merten. *Run-Time Optimization Architecture*. Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, August 2002.
`http://www.crhc.uiuc.edu/IMPACT/ftp/report/phd-thesis-matthew-merten.pdf`

[2] Morph: Late Code Modification.
`http://www.eecs.harvard.edu/morph/`

[3] Matthew C. Merten and Michael S. Thiems. An Overview of the IMPACT X86 Binary Reoptimization Framework. *IMPACT Technical Report*, May 1998.
`http://www.crhc.uiuc.edu/IMPACT/ftp/report/impact-98-05.binary.pdf`

[4] M. C. Merten. *A Framework for Profile-Driven Optimization in the IMPACT Binary Reoptimization System*. Master's thesis, 1999.
`http://citeseer.ist.psu.edu/merten99framework.html`

[5] B.S. Christopher Neith George. *A Framework for Install-Time Optimization of Binary Dynamic-Link Libraries*. Master's thesis, University of Illinois at Urbana-Champaign, 1997.
`http://www.crhc.uiuc.edu/IMPACT/ftp/report/ms-thesis-christopher-george.pdf`

[6] HeavenTools. PE Explorer: Debug Info Viewer, Relocation Viewer, Strip Tools and TimeDateStamp Adjuster.
`http://www.pe-explorer.com/peexplorer-tour-more-tools.htm`

[7] M. S. Thiems. *Optimization and Executable Regeneration in the Impact Binary Reoptimization Framework*. Master's thesis, 1998.
`http://citeseer.ist.psu.edu/thiems98optimization.html`

[8] Robert S. Cohn, David W. Goodwin, and P. Geoffrey Lowney. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), 1998.
`http://research.compaq.com/wrl/DECarchives/DTJ/DTJS01/DTJS01HM.HTM`

[9] Robert S. Cohn, David W. Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An Optimizer for Alpha/NT Executables. *USENIX Windows NT Workshop*, August 1997.
`http://citeseer.ist.psu.edu/lowney97spike.html`

[10] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining Emulation and Binary Translation. *Digital Technical Journal*, Vol. 9(No. 1), January 1997.
`http://research.compaq.com/wrl/DECarchives/DTJ/DTJP01/DTJP01PF.PDF`

[11] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A Profile-Directed Binary Translator. *IEEE Micro*, 18(2):55 – 64, March 1998.

[12] Anton Chernoff and Ray Hookway. DIGITAL FX!32: Running 32-bit x86 Applications on Alpha NT. *Proceedings of the USENIX Windows NT Workshop*, August 1997.
`http://www.usenix.org/publications/library/proceedings/usenix-nt97/full_papers/chernoff/chernoff.pdf`

[13] Sheldon Lobo. The Sun Studio Binary Code Optimizer, November 2005.
`http://developers.sun.com/sunstudio/articles/binopt.html`

[14] Charm Home Page.
`http://rogue.colorado.edu/Charm/`

[15] Pin - A Dynamic Binary Instrumentation Tool.
`http://rogue.colorado.edu/Pin/`

[16] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. SUIF: A Parallelizing and Optimizing Research Compiler. *Technical Report: CSL-TR-94-620*, 1994.
`http://citeseer.ist.psu.edu/context/315105/0`

[17] *Etch: Instrumentation and Optimization of WIN32/Intel Executables.*
http://etch.cs.washington.edu/

[18] Valgrind Technical Documentation.
http://www.valgrind.org/docs/manual/tech-docs.html

[19] Qemu Internals.
http://www.qemu.org/qemu-tech.html

[20] Code Compaction with aiPop for C16x/ST10, 2006.
http://www.absint.com/aipop/slides/

[21] *IA-32 Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 2004.
http://www.intel.com/design/pentium4/manuals/index_new.htm

[22] *IA-32 Intel® Architecture Optimization Reference Manual*, 2004.
http://www.intel.com/design/pentium4/manuals/index_new.htm

[23] *AMD Athlon™ Processor x86 Code Optimization Guide*, February 2002.
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf

[24] Jon "Hannibal" Stokes. The Pentium: An Architectural History of the World's Most Famous Desktop Processor, July 2004.
http://arstechnica.com/articles/paedia/cpu/pentium-1.ars/1

[25] Agner Fog. *The microarchitecture of Intel and AMD CPUs*, 2006.
http://www.agner.org/optimize/

[26] Agner Fog. *Instruction tables*, 2006.
http://www.agner.org/optimize/

[27] Agner Fog. *Optimizing subroutines in assembly language*, 2006.
http://www.agner.org/optimize/

[28] Agner Fog. *Optimizing software in C++*, 2006.
http://www.agner.org/optimize/

[29] *IA-32 Intel® Architecture Software Developer's Manual, Volume 2A, 2B: Instruction Set Reference*, 2004.
http://www.intel.com/design/pentium4/manuals/index_new.htm

[30] Christian Ludloff. IA-32 architecture, 2006.
http://www.sandpile.org/ia32/index.htm

[31] Summer 2004 Laboratory Notes, 2004.
http://courses.ece.uiuc.edu/ece390/books/labmanual/index.html

[32] Eric Rotenberg, Steve Bennett, and Jim Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. *International Symposium on Microarchitecture*, 1996.
http://citeseer.ist.psu.edu/rotenberg96trace.html

[33] CPU cache - Wikipedia.
http://en.wikipedia.org/wiki/CPU_cache

[34] Hans de Vries. Looking at Intel's Prescott die, Part II, April 2003.
http://chip-architect.com/news/2003_04_20_Looking_at_Intels_Prescott_part2.html

[35] klog <klog@promisc.org>. Backdooring Binary Objects. *Phrack Magazine*, Volume 0xa(Issue 0x38), May 2000.
http://www.phrack.org/archives/56/p56-0x09

[36] BFD Library manual.
http://sourceware.org/binutils/docs-2.17/bfd/index.html

[37] Gianfranco Bilardi and Keshav Pingali. The Static Single Assignment Form and its Computation. *Cornell University Technical Report*, July 1999.
http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/ssa.ps

[38] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451 – 490, October 1991.
http://citeseer.ist.psu.edu/cytron91efficiently.html

[39] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in Linear Time. *SIAM Journal on Computing*, Volume 28(Issue 6):2117 – 2132, June 1999.
http://portal.acm.org/citation.cfm?id=323341

[40] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. Linear-Time Pointer-Machine Algorithms for Least Common Ancestors, MST Verification, and Dominators. *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 279 – 288, 1998.
http://citeseer.ist.psu.edu/buchsbaum98lineartime.html

[41] Dov Harel. A linear time algorithm for finding dominators in flowgraphs and related problems. *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 185 – 194, May 1985.

[42] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A Simple, Fast Dominance Algorithm. *Software Practice and Experience*, 2001.

[43] Loukas Georgiadis, Robert E. Tarjan, and Renato F. Werneck. Finding Dominators in Practice. *Journal of Graph Algorithms and Applications*, 10(1):69–94, 2006.
http://www.emis.de/journals/JGAA/accepted/2006/
GeorgiadisTarjanWerneck2006.10.1.pdf

[44] G. Ramalingam. On Loops, Dominators, and Dominance Frontiers. *ACM SIGPLAN Notices*, 35(5):233 – 241, May 2000.

[45] David Eppstein. ICS 161: Design and Analysis of Algorithms.
http://www.ics.uci.edu/~eppstein/161/960220.html

[46] Fudan University, Shanghai. *Design and Analysis of Algorithms*, 2005.
http://www.cs.ust.hk/~rudolf/Courses/Alg_ug_06w/Resources/Script/index.html

[47] Stephen Alstrup, Inge Li Goertz, Theis Rauhe, Mikkel Thorup, and Uri Zwick. Union-Find with Constant Time Deletions. *Lecture notes in computer science*, 2005. ISSN 0302-9743.
http://www.springerlink.com/index/9J758DETC9F3X66J.pdf

[48] R. Cohn and P. G. Lowney. Hot Cold Optimization of Large Windows/NT Applications. *MICRO29*, pages 80 – 89, December 1996.

[49] Xianglong Huang, Brian T Lewis, and Kathryn S McKinley. Dynamic code management: improving whole program code locality in managed runtimes. *Proceedings of the 2nd international conference on Virtual execution environments*, pages 133 – 143, 2006. ISSN 1-59593-332-6.
http://www.cs.utexas.edu/users/xlhuang/dcm.pdf

[50] Cliff Young, David S. Johnson, Michael D. Smith, and David R. Karger. Near-optimal intraprocedural branch alignment. *Proceedings of the ACM SIGPLAN 1997 Conference on Programming language design and implementation*, pages 183 – 193, 1997. ISSN 0362-1340.
http://theory.lcs.mit.edu/~karger/Papers/pldi96-final.ps.gz

[51] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN 1990 Conference on Programming language design and implementation*, pages 16 – 27, 1990. ISSN 0362-1340.
http://www.cs.rice.edu/~keith/512/Lectures/12PettisHansen.pdf

[52] *Intel® Processor Identification and the CPUID Instruction*, June 2001.
http://developer.intel.com/design/xeon/applnots/241618.htm

[53] Red-black tree - Wikipedia.
http://en.wikipedia.org/wiki/Red-black_tree

[54] Daniel Dominic Sleator and Robert Endre Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652 – 686, 1985. ISSN 0004-5411.
http://www.cs.princeton.edu/courses/archive/fall05/cos528/handouts/splay.pdf

Index

T

U

V