MSE | MASTER OF SCIENCE
IN ENGINEERING

BERKELEY LAB
Lawrence Berkeley National Laboratory

Cache-Friendly Profile Guided Optimization

M.Sc. Thesis

Baptiste Wicht
baptiste.wicht@gmail.com

Professor

Frédéric Bapst, *EIA-FR*
frederic.bapst@hefr.ch

Expert

Noé Lutz, *Google*
noe.lutz@gmail.com

Supervisors

Roberto Agostino Vitillo, *LBNL*
ravitillo@lbl.gov

Paolo Calafiura, *LBNL*
pcalafiura@lbl.gov

Department: Information and Communications Technology
Field: Computer Science
Keywords: Compilers, Profile-Guided Optimization, Cache optimization, Hardware Performance Counters, GCC
Project Period: 17.09.2012 - 08.02.2012

Preface

This thesis is submitted in partial fulfillment of the requirements for Master of Science in Computer Science at the University of Applied Science, Fribourg for Baptiste Wicht. This project took place in Lawrence Berkeley National Laboratory, California for five months.

Acknowledgments

Although this thesis results of the compilation of my own efforts, I would like to acknowledge and express my gratitude to the following people for their valuable time and assistance, without whom the completion of this project would not have been possible:

- The Lawrence Berkeley National Laboratory and especially Paolo Calafiura who gave me the opportunity to do this very interesting project at LBNL.
- Roberto Vitillo (LBNL), my project supervisor, for his help and support during the whole project.
- Frédéric Bapst (HES-SO) for all his precious advice and the time he devoted to review the numerous pages of this report.
- Dehao Chen (Google) for all his answers about the Google AutoFDO patch and for providing me with his own versions of the AFDO profiles which have proved more than useful to test the converter.
- David Levinthal (Google) for his help for using and understanding Gooda and for fixing the bugs found during this project.
- Stephane Eranian (Google) for his help for installing and configuring Gooda and `perf`.
- All the participants of the "Friday Software Quality Lunch" for the excellent lunches, the fun discussions and the bugs avoided.
- The Hirschmann Stipendium for their scholarship which helped me a lot during my Master.

Abstract

Modern compilers are providing advanced optimizations: Inlining, Loop Transformations, Code Layout Optimizations, etc. These techniques base their decisions on static information collected from the program source code. This generally gives good results. However, by knowing how the program behaves at run-time, the compiler can do even better. This technique is called Profile Guided Optimization (PGO).

At the time of this writing, compilers are only supporting instrumentation-based PGO. This technique proved effective in bringing very good performance. However, few projects use it, due to its complicated dual-compilation model and the high profiling overhead. Sampling Hardware Performance Counters overcomes these drawbacks. This approach has a much lower overhead, which allows to collect profiles in production environments, that in turn generate accurate profiles.

This paper focuses on the GNU Compiler Collection (GCC) compiler. It provides a new open-source toolchain allowing sampling-based PGO. The toolchain is based on the perf profiler, the Gooda analyzer, AutoFDO and GCC.

By using LBR-sampling, the generated profiles are very accurate. This solution achieves an average of 83 percent of the gains obtained with instrumentation-based PGO and 93 percent on C++ benchmarks. The profiling overhead is only 1.06 percent on average, whereas instrumentation incurs a 16 percent overhead on average.

Memory if one of the most important bottlenecks in modern applications. To help improve the memory usage, the proposed toolchain also brings information about Memory Latency into GCC, creating a memory-profile for PGO. A Loop Fusion optimization taking advantage of the new Memory Latency information to choose the most interesting loops to merge has been developed for GCC.

Contents

Chapter 1

Introduction

For software developers and companies, the runtime of programs has always been a very important topic. It is especially true for programs working on the cloud. When you pay for each minute they run, it becomes very important to optimize a program as much as possible.

To know where the main areas for optimization are, programmers use tools called profilers. They run the program using this tool and it collects statistics about which functions are the most time-consuming, what instructions are executed the most, etc. Then, the developer can use these results to find the hotspots to optimize.

This process is time-consuming, but it can be automated. Modern compilers include an optimization technique called Profile Guided Optimization (PGO). The compiler uses a profile of the program execution to perform optimization techniques. This is generally more interesting as it depends on real profile instead of static estimations calculated by the compiler.

PGO has proved effective in several domains, but has not been widely adopted, due to its complicated dual-compilation model and its high overhead. One objective of this project is to make PGO more convenient by using Hardware Performance Counters provided by modern processors.

These tools and techniques help producing programs consuming less cycles of the processor. However, the bottleneck of modern applications is often the main memory. The main memory is orders of magnitude slower than the processor itself. Sometimes, the processor wastes its time waiting from data coming from the memory.

The second objective of this project is to try to provide a solution to improve memory usage by adding memory profile to PGO.

1.1 Context

At the time of this writing, all compilers supporting PGO collect the profile by using instrumentation. A specific version of the application is created with new instructions inserted at strategic locations to collect a profile. During the execution, the profile is generated to a file. Once the application has been executed, the collected profile is used to optimize the program further during a second compilation.

This approach has several drawbacks:

- The instructions added to the program slow it down. An instrumented binary can be more than one order of magnitude slower than the final version. Such a binary cannot always be used in production. If used on wrong data input, it can happen that the generated profile is not similar enough to the production pattern, resulting in a loss a performance.

- The process is not very convenient. It is necessary to compile the program a first time with special flags, then to execute it and finally compile it again with the generated profile. Large projects can take hours to compile. Compiling it twice is not always an option.

- The instrumentation instructions can alter the quality of the collected profile.

- Only a few different information can be collected. Indeed, it is not possible with this approach to collect information about cache-misses, page-faults or branch-prediction misses. The reason is that the instrumentation instructions are just incrementing simple counters to compute code coverage.

Modern processors are providing Hardware Performance Counters (HPC). These counters are automatically managed by the processor and are read-only accessible from the software part. There is a broad range of events, for instance, instructions retired, cache misses, branch mispredictions, the number of branches executed, etc.

These events can be used to perform sampling by collecting their values at some interval. The interval is expressed in number of events. Each time the interval is reached, the current instruction is saved with the current state of the counters. These data together form a sample. At the end of the program execution, a valid profile of the application is generated, containing a list of samples. Sampling can be performed on any binary.

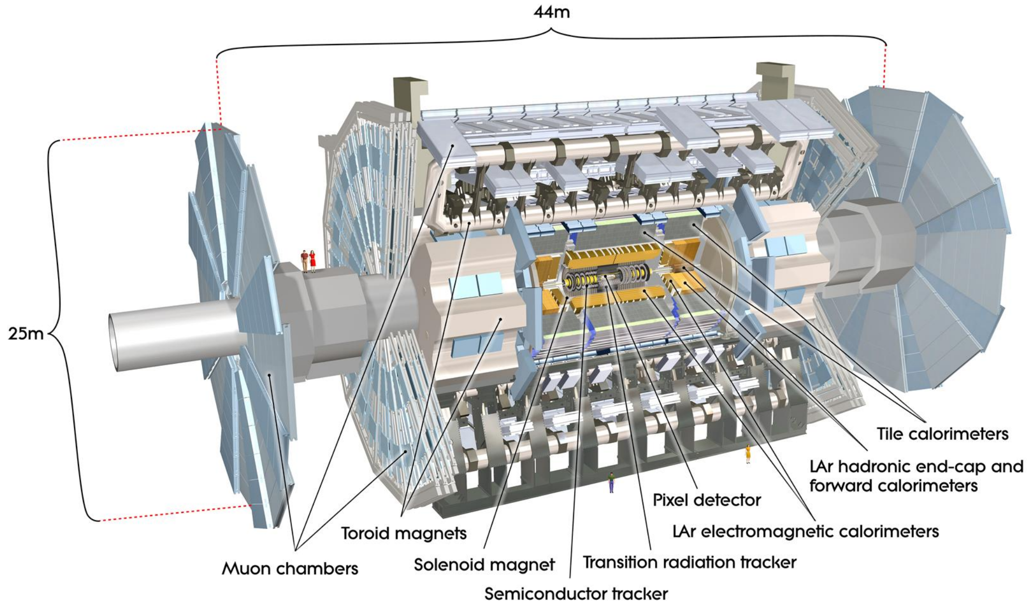

44m
25m
Tile calorimeters
LAr hadronic end-cap and forward calorimeters
Pixel detector
LAr electromagnetic calorimeters
Toroid magnets
Muon chambers
Solenoid magnet
Transition radiation tracker
Semiconductor tracker

Figure 1.1: Computer generated view of the ATLAS detector

Source: ATLAS Experiment ©2012 CERN

1.1.1 ATLAS

ATLAS is one of the four experiments at the Large Hadron Collider (LHC), the world's most powerful particle accelerator, located at the CERN Laboratory in Geneva, Switzerland. The LHC has been built to provide suitable conditions to explore new frontiers in High Energy Physics (HEP) producing proton-proton collisions at a design center of mass energy of 14 TeV. ATLAS acts as a huge digital camera composed of millions of electronic channels registering the characteristics of the particles emerging from the interactions with very high precision. It has a standard collider experiment structure of concentric cylinders consisting of tracking detectors, calorimeters and muon spectrometer, centered around the collision region. The tracking device, immersed in a 2 tesla magnetic field, measures the charge and momentum of charged particles. The outermost cylinder is the muon spectrometer, a detector which measures the momentum and charge of muons, particles capable of traversing the full volume of ATLAS without being absorbed.

An internal view of the ATLAS detector is presented in Figure 1.1.

ATLAS is involved in the discovery of an Higgs-like boson, in July 2012. The Higgs boson is an elementary particle in the Standard Model of particle physics. The Higgs boson has been predicted a long time ago, but was never observed before. The discovery should be confirmed, or infirmed, in 2013.

ATLAS produces a humongous amount of data during a LHC experiment. These data are then analyzed by the ATLAS software, running on a computer grid. 3.2 Petabytes of data are collected and analyzed each year. For such an amount of data, the processing performance of the software is crucial. Several analyses are limited by the amount of running time that is available for the software.

For several years, the ATLAS software team at LBNL has been working on improving the efficiency of the software. This project is part of their work to use PGO to improve ATLAS software performance.

1.2 Goals

The main goal of this project is to improve the use of PGO by using Hardware Performance Counters instead of instrumentation in modern compilers.

The first objective of the project is to see how Sampling-based PGO can be integrated in a compiler. It will be necessary to choose a compiler to work with. Then, its PGO features will be studied in detail. Performance Monitoring Events will also have to be studied to know how to use them to perform accurate sampling and which events are the most interesting to integrate in the solution. Special attention should be paid to the conversion of a sampling-based profile to an edge profile, generally generated by a precise instrumentation-based profiling.

Then, the second objective will be to bring more Hardware Performance Events into this compiler, for instance Data Cache Misses. Finally, it would be necessary to find a way to use this new information inside the compiler to improve a program efficiency. This can be done by using the information in an existing pass to help decision or by implementing a new optimization pass using this information. Several events should be considered potential candidates before choosing at least one of them for implementation.

Finally, it will be necessary to verify if this approach is efficient enough (in both terms of profiling time and of efficiency of the generated program) to be used in production. The performance gains of this new approach should be compared to the gains brought by instrumentation-based PGO.

1.3 Related Work

Instrumentation-based PGO is known and used for a long time. On the other hand, researches on the usage of sampling to perform PGO only started recently.

In 2008, Roy Levin, Ilan Newman and Gadi Haber [Levin2008] proposed a solution to generate edge profiles from instruction profiles of the instruction retired hardware event for the IBM FDPR-Pro, post-link time optimizer. This solution works on the binary level. The profile is applied to the corresponding basic blocks after link-time. The construction of the edge profile from the sample profile is known as a Minimum Cost Circulation problem. They showed that this can be solved in acceptable time for the SPEC benchmarks, but this remains a heavy algorithm.

Soon after Levin et al., Vinodha Ramasamy, Robert Hundt, Dehao Chen and Wenguang Chen [Ramasamy2008] presented another solution of using instruction retired hardware events to construct an edge profile. This solution was implemented and tested in the Open64 compiler. Unlike the previous work, the profiles is reconstructed from the binary using source position information. This has the advantage that the binary can be built using any compiler and then used by Open64 to perform PGO. They were able to reach an average of 80 percent of the gains that can be obtained with instrumentation-based PGO.

In 2010, Dehao Chen et al.[Chen2010] continued the work started in Open64 and adapted it for GCC. In this work, several optimizations of GCC were specially adapted to the use of sampling profiles. The basic block and edges frequencies are derived using a Minimum Control Flow algorithm. In this solution, the Last Branch Record (LBR) precise sampling feature of the processor was used to improve the accuracy of the profile. Moreover, they also used a special version of the Lightweight Interprocedural Optimizer (LIPO) of GCC. The value profile is also derived from the sample profile using PEBS mode. With all these optimizations put together, they were able to achieve an average of 92 percent of the performance gains of instrumentation-based Feedback Directed Optimization (FDO).

More recently, Dehao Chen (Google) released AutoFDO[1] (AFDO). It is a patch for GCC to handle sampling-based profiles. The profile is represented by a GCOV file, containing function profiles. Several optimizations of GCC have been reviewed to handle more accurately this new kind of profile. The profile is generated from the debug information contained into the executable and the samples are collected using the perf tool. AutoFDO is especially made to support optimized binary. For the

[1] http://gcc.gnu.org/ml/gcc-patches/2012-09/msg01941.html

time being, AFDO does not handle value profiles. Only the GCC patch has been released so far, no tool to generate the profile has been released.

To the best of our knowledge, uses of other hardware performance events with PGO has not been attempted so far.

1.4 Structure of this document

This document is divided into 10 chapters, including this introduction.

The next chapter covers the preliminary analysis of the project about the different concepts as well as the choice of a compiler. Chapter three treats of Performance Counters, with special care for the Intel® Ivy Bridge Counters. Chapter four covers GCC, its optimization techniques, its PGO features and its architecture. Then, the following chapter covers the different optimizations that are performed by compilers to optimize the usage of the main memory. The next two chapters are covering the implementation that has been performed during this project. The first one being the support for PGO by sampling and the addition of Cache Misses information into GCC. The second is the development of a Loop Fusion pass into GCC. After these two chapters, the usage of performance counters in compilers is discussed in detail. Then, the challenges encountered during this project are detailed. Finally, a conclusion of the project is drawn.

After the main chapters, several appendices are available. The content of the archive provided with this report is described in the first one. Then, the installation procedure of the tools developed during this project is explained in detail. The third appendix contains a list of the Performance Events available on an Intel® Ivy Bridge microarchitecture. The last appendix list all the optimization passes performed by GCC.

Chapter 2

Analysis

This chapter covers the preliminary analysis of this project. The goal of this phase is to collect enough information to take decisions in later phases. This is especially important to decide on which compiler this project will focus.

This chapter starts by studying the different compilers (GCC and CLang have been considered). Then, the Hardware Performance Counters and the `perf` profiler are covered in detail. The memory and cache organization of modern processors are also studied. Finally, Generic Optimization Data Analyzer (Gooda) is analyzed.

2.1 Compilers

A compiler is a tool transforming a source file written in a specific programming language into an executable that is understandable by the target machine.

More than just compiling a program into an executable, modern compilers are applying a broad range of optimization to make it as fast as possible (with the condition that it remains correct). Generally, this is the main criteria used to compare different compilers. The optimization process is the most complex and time-consuming task process of a compiler. Most optimization techniques are known as NP, thus compilers use heuristics to optimize programs in reasonable time.

Most modern compilers have separated front ends and back ends with a common middle-end. This architecture allows to implement only once all the optimizations and to profit from all the existing back ends for each front end. This is achieved by using a specific Intermediate Representation to communicate from the front end to the back end.

Optimization can be done at several levels:

1. **Local**: Optimize a specific Basic Block. This is, by far, the simplest optimizations. Since basic blocks have no control flow, analysis is very easy to perform. Some optimization techniques are even performing only on one instruction at a time (e.g. Constant Folding).

2. **Intraprocedural**: Also called global optimizations. Optimize together all the basic blocks of a function. It takes the control flow graph of the function and follows it during the optimization. This generally involves data-flow analysis algorithm that can be quite expensive. This is where most of the optimizations are performed.

3. **Interprocedural**: Optimize through the whole call-graph of the program. This optimization is based on the call graph of the whole program. It can be very expensive to perform. Some optimizations can be very complex to implement at this level.

There are two main families of optimization:

- **Static Optimization**: Optimization based on the information collected or predicted by the compiler in the source files.

- **Dynamic Optimization**: Optimization done in a two-phase manner. The application is compiled once. It is executed and data about the run are collected. The application is then compiled a second time using the collected data to improve optimization.

Generally, the second set of optimization is the same as the first, but uses different data. When no profiling data is available, the compiler has to guess some of the information (e.g. branch frequency). This is done by using a set of heuristics on the static representation of the program.

If the execution reflects accurately the general usage pattern of the application, the second version of the application should be faster than the first. Nevertheless, if the usage pattern is not accurate, it can make the program slower in the general case. Where static optimization tries to improve the execution for all cases, dynamic optimization improves it for a specific case. Some programs profit a lot from it whereas other programs are not really improved.

Dynamic Optimization, also called Profile Guided Optimization (PGO), or Feedback Directed Optimization (FDO) is the optimization that can profit from the usage of Hardware Performance Counters. Section 2.1.1 gives more details about PGO.

2.1.1 Profile Guided Optimization

PGO uses information collected during the execution of a program to optimize it further. Several optimization techniques can take advantage from these new data. For instance, functions calling each other frequently can be put close in the generated code to reduce Instruction Cache misses. Branches can be reordered to match their frequency to avoid branch misprediction. Loops working on arrays causing Data Cache misses can be improved to make better use of the cache.

There are two ways to collect profile data at runtime:

- **Instrumentation** This is the most common way. The compiler creates a special version of the application with new instructions inserted at specific locations that output data to a file. During the program's execution, these new instructions collect the necessary data to fill the profile.

- **Sampling** The application does not have to be specially compiled. During the execution, a special program, called a profiler, collects data from the program to fill a profile. The data can be collected using call-stack sampling or Hardware Performance Counters sampling.

The complete process of Instrumentation is described in Figure 2.1.

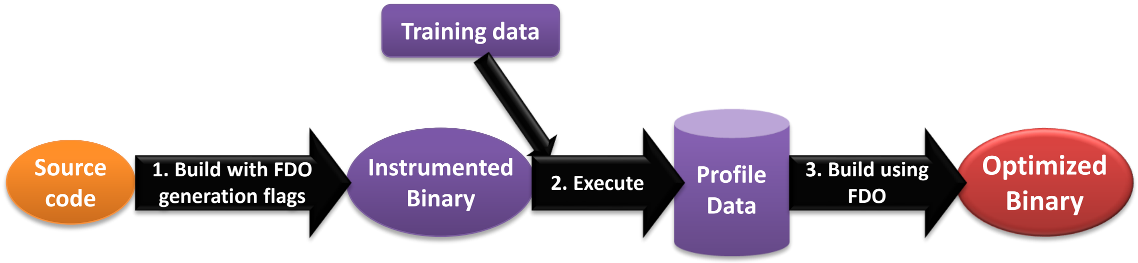


Figure 2.1: PGO with Instrumentation Process

Source: [Papaux2012]

Due to the weight of this process and the disadvantages of instrumentation, a lot of projects (even CPU intensive projects) do not use this feature. Sampling would perhaps give these projects an opportunity to test and use this feature.

The first approach is very precise regarding to instructions executed, but it has an important overhead on the program execution time. Some applications are running orders of magnitude slower when instrumented. On the other hand, sampling is not as accurate as instrumentation, but is much lighter. With this small overhead, sampling can easily be used on production, generating a profile on the most accurate input. As sampling does not need a specifically compiled executable, a profiler can generally be launched on an already running program. It is not necessary to profile the whole execution, keeping in mind that the profiled period should be relevant. Moreover, sampling can be used with a broader range of events whereas instrumentation is much more limited. For instance, it is not possible to collect cache misses statistics by Instrumentation.

Another difference is that Instrumentation is portable, whereas Sampling is not always because each processor supports a different set of events.

At the time of this writing, no widely-used compiler has support for sampling-based PGO.

2.2 GCC

The GNU Compiler Collection (GCC) [1] compiler is a free and open source compiler project started in 1987 for the GNU Operating System. It is an important part of the GNU Toolchain, along with GDB and Make for instance. It supports a wide range of programming languages: C, C++, Java, Google Go, Fortran, etc.

GCC is maintained by the GNU Project and is still under active development. At the time of this writing, the latest stable version is the 4.7.2 which has been released on September 20th, 2012.

The GCC front end parses the source code into an Abstract Syntax Tree (AST) (represented in a language-independent tree structure, GENERIC). It is then passed to the middle end in GIMPLE format. GCC uses its own Intermediate Representation (IR): GIMPLE (mid-level) and Register Transfer Language (RTL) (low-level). Optimizations are applied directly to the IRs. Finally, the back end generates an executable understandable by the target.

[1] `http://gcc.gnu.org/`

2.2.1 Optimization

GCC implements a very broad range of optimization techniques. It supports several levels of optimization:

- **-O0** No optimization are performed. This is the default level.
- **-O1** Only basic optimizations are performed. No time-consuming optimization is performed at this level.
- **-O2** Even more optimization passes are enabled. These optimization passes do not involve space-speed trade off. This is the recommended level for production binary. GCC itself is compiled with O2.
- **-O3** Turns on optimization that can greatly increase the size of the generated code. Due to this increase in size, O3 is not always faster than O2.
- **-Ofast** Disregard strict standard compliance. Activate all the O3 optimization passes and activate techniques that are not standard compliant.
- **-Os** Optimize for size. It enables all the O2 options that do not increase code size. It also performs further transformations to reduce code size.

Each level turns on the options of the previous level. Each level turns on a set of optimization passes that can also be enabled or disabled separately.

GCC has long been criticized for its lack of powerful Interprocedural Optimization, but this has been improved since then and GCC is now able to perform it fairly well.

GCC also supports Link-Time Optimization (LTO). When LTO is performed, very few optimization passes are made at compile-time and the generated object file contains GIMPLE representation of the file. Then, at link-time, the object files are put together and optimized as if it was a single source file. This can generate faster executable. This process is generally much slower than performing standard compilation.

GCC supports very modern processor features (e.g. SSE4.2 or AVX). It can also automatically vectorize or parallelize some loops.

2.2.2 Profile-Guided Optimization

GCC uses GCOV to collect the profile during the execution. A profile indicates which basic blocks are often executed and which are not. It is collected by instrumenting the program, adding assembly instructions to the program. These instructions increment counters each time execution flows through this place. GCC does not instrument each Basic Block, instead it instruments the arcs between them. Moreover, to limit the overhead, it does not count all the arcs, but uses an optimal placement. This placement is based on a spanning tree of the Control Flow Graph (CFG). The counters can then be propagated through the arcs to have a complete profile.

There are some limitations to take into account when applying PGO. These limitations are made to have matching CFG. Without these limitations, the CFG would be different and so it will not be possible to match the profiling data to the program being compiled. First, the same optimization options must be used for the first and the second compilation. Moreover, the pass adding instrumentation instructions and the one using the results must be at the same position in the sequence of optimization passes.

Several passes take advantage of profiling data, e.g. basic block reordering, function reordering and register allocation. When no profile data is available, the frequencies of the branches are predicted using static heuristics. The optimization passes do not perform differently whether profile information is available or not.

2.3 LLVM CLang

CLang is a compiler for the C, C++ and Objective-C programming languages, part of the Low Level Virtual Machine project[2]. It is a relatively new project, started in 2000, with version 1.0 released in 2003. The goal of this project is to provide a modular architecture with reusable components.

CLang is the C and C++ front end. It uses the LLVM Core component that provides optimizer, code generation, etc.

CLang has been designed to be compatible with GCC. Most of the options are the same, but CLang does not support all the options of GCC. GCC supports several non-standard extensions that CLang does not support.

The CLang front end parses the source into an AST. After all the language-dependent

[2] http://llvm.org/

analysis have been performed on the AST, CLang translates it into an Intermediate Representation, the LLVM IR. The representation is passed to the LLVM Core. The optimization of the program is performed by the LLVM Core. After all the optimization passes, the back end generates machine code for the program.

CLang claims to be faster and more memory-efficient than GCC[3]. The diagnostics provided are generally of better quality than with GCC.

2.3.1 Optimization

There are two ways to configure the optimization passes performed by CLang. The most powerful is to use the `opt` tool (the LLVM Optimizer) on the generated LLVM IR code. The other one, and the more common, is to set the optimization levels to CLang:

- **-O0** No optimization are performed.
- **-O1** Only basic optimizations are performed. No time-consuming optimization is performed at this level.
- **-O2** Even more optimizations are performed. These techniques do not involve space-speed trade-off and do not slow down the compilation too much.
- **-O3** Turns on optimization that can greatly increase the size of the generated code.
- **-O4** Turns on LTO.
- **-Os** Optimize for size. It enables all the O2 optimization options that do not increase code size.
- **-Oz** Optimize for size even further. It enables all the Os optimizations and activates other optimizations that reduce the code size even further.

Each level turns on the options of the previous level.

LLVM supports optimization at each level. It supports modern processor features (e.g. SSE4 or AVX). It can also automatically vectorize some loops. However, the LLVM Loop Vectorizer is less advanced than GCC's Vectorizer.

[3] `http://clang.llvm.org/features.html`

2.3.2 Profile-Guided Optimization

CLang uses `llvm-prof` to collect code coverage information. Like GCC, CLang uses instrumentation to collect these statistics.

Unlike GCC, LLVM Core supports several kinds of profiling:

- **Edge Profiling** Counters are added to all edges of the CFG. This is the way that adds the most overhead to the generated program.
- **Optimal Edge Profiling** Use a better placement of the counters to have as few counters as possible. The counters can then be propagated to have a full coverage report.
- **GCOV Profiling** The generated counters are partially compatible with GCOV. Nonetheless, they are only usable for reporting not for use in GCC.
- **Path Profiling** Add counters to all paths of the CFG. A path is a sequence of basic blocks representing a possible execution of the function.

CLang uses its own file format to store this profile.

At the time of this writing, there are no passes using the profile information. It is only read during one pass and not used anymore. Moreover, the optimization passes must be disabled when PGO is enabled. It is planned to remove the actual profiling system that is not used anymore[4]. Nevertheless, a 2012 Google Summer of Code project has started to improve the PGO support in CLang[5].

2.4 GCC or CLang ?

It is not possible to perform a complete study of both compilers at the same time. It is necessary to choose one.

Both compilers have their strengths and weaknesses. They are compared in terms of architecture, maturity, documentation, optimization capabilities and PGO support.

[4] http://lists.cs.uiuc.edu/pipermail/llvmdev/2012-July/051745.html

[5] http://www.google-melange.com/gsoc/project/google/gsoc2012/alycm/17001

Architecture

The LLVM Architecture is highly modular. Each component is reusable and can be used easily in another application. Moreover, LLVM is written in C++ in an object-oriented fashion. On the other hand, GCC has an old architecture and is mostly written in an old fashioned C. It is much harder to dig into GCC than into LLVM. Nevertheless, the architecture of GCC is also separated into components, not really reusable, but it makes possible to perform changes at a specific place without having to wonder too much about the other locations.

Both compilers have a separated front end, back end architecture that makes it practical to add support for a new programming language or a new target.

Maturity

GCC is highly mature and is used daily to compile a broad range of applications. It is the compiler used to compile the Linux Kernel. On the other hand, CLang is much younger. However, it is supported by some major companies and it is now the default compiler of the FreeBSD Operating System. Both compilers are in constant evolution.

Documentation

GCC has been the leader in compilers for years. As a result, there are plenty of articles and books about it. The official GCC site contains a very complete documentation about its internals. LLVM internals are not as well documented.

For both compilers, a list of optimization passes is available. The source code of both compilers is well documented with complete explanations and links to reference articles about the algorithms used.

Optimization

Both compilers offer a broad range of optimization techniques. The optimizations are made in sequential passes, transforming the program or gathering information about it. GCC has some optimization techniques more advanced than CLang. In most of the cases, the generated binaries from GCC are faster than the equivalent CLang binary. Nonetheless, the difference is going less and less significant with the latest versions of CLang and sometimes the executables generated by CLang are faster than when compiled with GCC.

Profile-Guided Optimization

When it comes to PGO, GCC is much more advanced than CLang. CLang is able to produce profiling data and to annotate its program representation with these data.

Nevertheless, it does not use these data. On the other hand, GCC uses the profile in several passes (function reordering, modulo scheduling, etc. [Ramasamy2008]).

Conclusion

Although the modularity and clean architecture of CLang is a big advantage, its PGO support is not very developed. No pass uses the profile and the other optimizations must be disabled when using PGO. GCC supports several other optimization passes and the other optimization passes can be activated when doing PGO. Moreover, the GCC internals are well known.

For these reasons, it has been decided to use GCC for the next steps of the project. It will be harder to dig into the source, but the state of PGO in LLVM is not advanced enough for this project.

2.5 Hardware Performance Counters

Modern micro processors include a set of counters that are updated when a particular hardware event arises. This set of counters is managed by the Performance Monitoring Unit (Performance Monitoring Unit (PMU)) of the CPU. These events can then be accessed by an application. The contents of the counters can be dumped to the console at the end of the program execution. They can also be used for sampling. In this case, the PMU has to be configured to generate interrupts when a counter goes over a specified limit. At this point, the monitoring software can record the state of the system (Program Counter (PC)) and match it with the content of the counters. All these data will form a complete profile at the end of the execution. This form of sampling has a very low overhead (depending on the sampling interval).

The available events depend on the micro processor. The number of these events can vary a lot between models. For example, the Intel® Ivy Bridge processor has about 200 counters, whereas the ARM V7 Cortex-A9 processor has only 69 and where the PowerPC Power7 family has more than 1,500.

2.5.1 perf

`perf`[6] is a set of profiling tools for the Linux operating system. It gives access to hardware performance counters and abstract the differences between processors. It is

[6] https://perf.wiki.kernel.org/

based on the `perfs_event` system exported by the Linux Kernel. This subsystem is directly integrated in the Linux Kernel (since the 2.6.3.1 version) and gives access to the performance counters of the underlying processor. The `perf` tool is developed inside the kernel repository itself.

The `perfs_events` subsystem provides an abstraction of processor hardware events. Some of the events are grouped together so that they can be used transparently from one processor to another. As the available events are very different between different processors, the sets of common events is limited. Moreover, on some processor, it is even possible that some of these common events are not available. It is also possible to access directly the hardware counter by its processor name. This is useful if it is necessary to access a very specific hardware counter on a processor.

The events are divided into categories:

- **Hardware event** Events coming directly from the PMU (e.g. instructions, cpu-cycles, branch-misses, etc.)
- **Software event** Events coming from the Linux Kernel counters (e.g. context-switches, page-faults, etc.)
- **Hardware cache event** Events coming from the PMU, specific to the cache hierarchy (e.g. L1, Last Level Cache (LLC), TLB, etc.). These events are common to several processors. There are mapped to the specific hardware event if it exists.
- **Hardware breakpoint** Special events activated when a specific memory location is accessed. These breakpoints can be configured using `perf`.
- **Tracepoint event** Specific trace in the Linux Kernel that can be activated at run-time to debug this location (e.g. number of kmalloc or kfree).

Compared to other profilers, `perf` has a very small overhead on the program execution time, by using Hardware Performance Counters over sampling. This generally consists of just a small noise. Moreover, it has access to a very broad range of counters that no other profiler supports.

Event sampling captures only a fraction of the events. The interval of time can be configured, it is often between 1,000 and 1,000,000 events. The lower the interval, the higher overhead and the more accurate profile. With that property, the number of times an instruction has been executed is only a relative value, but if there are enough events, it is generally enough to have accurate aggregated profile.

2.5.2 Challenges

Using the data from perf in a compiler is not straightforward. The format is very different and even the way the data are collected is different.

Generally, profiling by instrumentation is done on edges whereas the samples are collected by instructions. The samples can then be aggregated for each basic block. Then, the edge frequencies have to be estimated from the basic block counters. Nonetheless, this transformation is far from trivial. Moreover, it is necessary to perform some balancing of the imprecisions to have accurate profiling information. It has been shown that it is not possible to transform instruction profiles into exact edge profiles in general [Probert1982]. Nevertheless, an algorithm has been defined in [Levin2008] and has already been used in [Ramasamy2008]. This algorithm, even if not optimal, provides a very good approximation of an edge profile. Hardware Performance Counters have been used in GCC by converting the perf profile into a GCOV profile file [Papaux2012]. However, this work has shown that the results are not totally accurate.

Another difficulty is about optimization passes and the state of the CFG. When a program is profiled, the only information available is the final program after all optimization passes. Nevertheless, when the PGO pass is activated, the CFG of the internal representation is not the same because not all the optimization passes have been performed on it. That makes it hard to match the information from the profile to the internal program representation.

2.6 Memory and Caches

An advantage of Hardware Performance Counters is the support of a large variety of counters, including counters for memory cache statistics. Thus, it is important to know the memory organization of modern microarchitecture.

Each cache works in the same way. When the processor needs a data from the memory, it first asks the cache. If the cache does not have the necessary data (it misses), it is fetched from higher cache levels into the cache and so on. When the last level of cache (LLC) misses, the data is fetched from main memory. When a data is fetched from main memory, a whole line of cache is filled. If only one byte is accessed, there will be more than eight bits fetched from the memory. This is done to improve the performances of the next accesses. If no other access to this cache line is made, there is no improvement. This is why spatial and temporal locality are very important.

Figure 2.2 shows the Memory Organization of an AMD Athlon™64 K8 Core as an example.

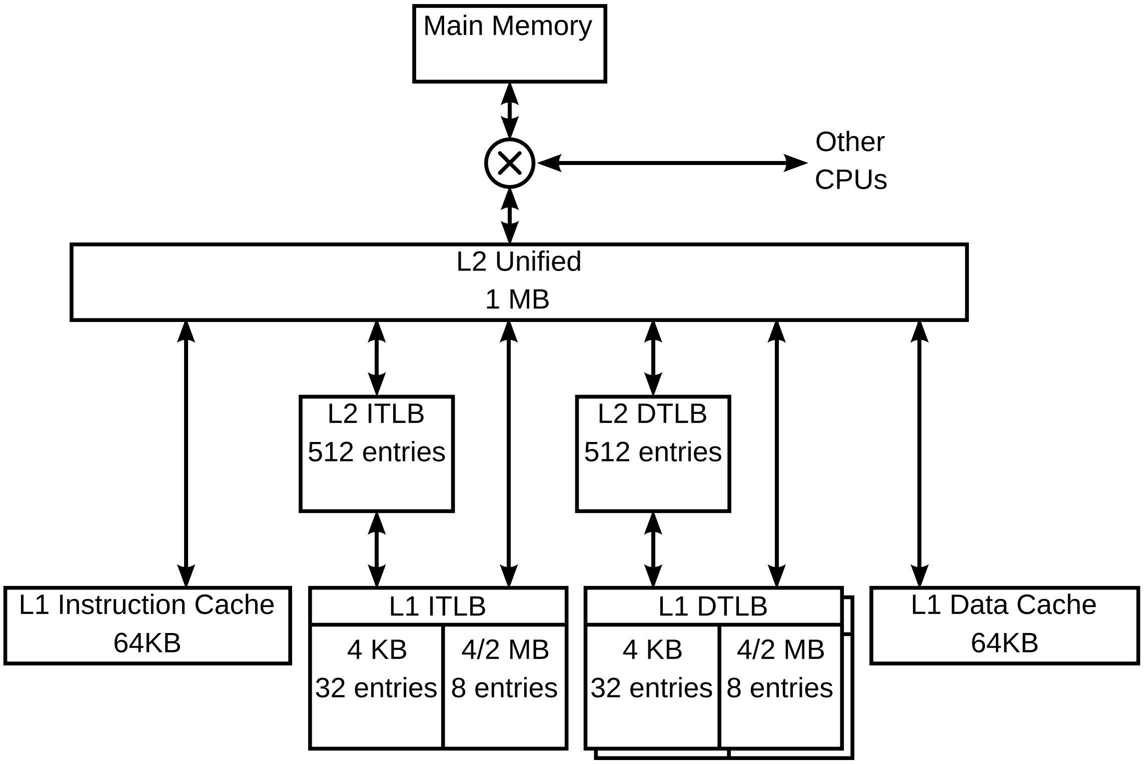


Figure 2.2: AMD Athlon™64 K8 Core Memory Organization

Source: Wikipedia

Even if there are several differences through processors, the example shows some important points. The example includes the four most common cache types:

- **Instruction Cache** When an instruction needs to be executed, it is read from the Instruction Cache. The cache holds many instructions.
- **Data Cache** The data cache stores recently accessed data from memory.
- **Instruction Transaction Lookaside Buffer** The ITLB is a buffer to speed up conversion of virtual-to-physical addresses for instructions. When a conversion is done, it is stored in the buffer to use again later.

- **Data Transaction Lookaside Buffer** The DTLB is a buffer to speed up conversion of virtual-to-physical data addresses.

A fundamental fact is that, the larger is the cache, the better is the hit rate, but the longer is the latency. To address this problem, most processors use multiple levels of cache. The lower level of the cache (generally L1) is checked first. If it hits, the data is directly given to the processor. If it misses, the next level is checked, and so on. When the last level of cache (LLC) misses, the data is fetched from main memory. Processors are using as many as three levels of cache.

On processors with several cores, some caches are local to each core (one different cache for each core) and some are shared.

In the example, there is a unified L2 cache used by all the sub caches. In some architectures, there is separate L2 caches and there can also be higher levels of caches.

All the caches are very important to speed up programs. Indeed, a L2 cache is typically an order of magnitude faster than main memory and a L1 cache is an order of magnitude faster than L2. The faster the cache is, the smaller it is, so L1 cache are generally very small (up to 128 KB), whereas L2 caches up to 16MB are available and main memory can hold thousands of GB.

Some modern processor includes also a micro-instruction cache (e.g. Intel® Sandy Bridge). This cache stores the instructions as they are decoded. An assembly instruction can be decoded into several micro-instructions. It is sometimes referred a L0 cache, but it is not related to higher levels of memory.

Moreover, the registers of a processor are also sometimes considered the first level of cache. It is indeed the fastest data storage on a processor, but it is not directly related to main memory, it has to be scheduled coarsely, generally by the compiler.

2.7 Gooda

Gooda is an analyzer for Performance Monitoring Unit events[7]. It consists of data collection scripts. These scripts are using `perf` to profile the application. Then Gooda analyzes the generated data using a cycle accounting methodology and creates spreadsheets of the results. It also provides a web visualizer to study the data and

[7] https://code.google.com/p/gooda/

view the CFG and Call Graph of the sampled application. Gooda manages several views for each function: the source view, the CFG view and the assembly view.

Gooda is an open source project. It is developed in a collaboration between Google and the LBNL. The author of the analyzer, David Levinthal (Google), has been engineer at Intel, specialized in Hardware Performance Events.

The goal of Gooda is to lower the required expertise for the user to profile its application with Performance Monitoring Events.

Gooda performs cycle accounting [Levinthal2008]: it uses the samples to calculate cycles. Gooda proposes a decomposition of the cycles into different groups of events. The cost (penalty paid in cycles for an event) of the events is taken into account to compute the values of the cycles. Gooda performs a system wide analysis. All the running processes, including the kernel itself, are profiled.

The main groups of events proposed by Gooda for High Performance Computing are:

- Load Latency. Cycles spent waiting for a data.
- Instruction Starvation. Cycles spent waiting for instructions.
- Branch Misprediction. Cycles wasted by branch misprediction.
- Function Call Overhead. Cycles spent in calling functions.

Gooda manages a large set of events. However, processors have limited counters. When there are more events than counters, multiplexing is used with the events managed in a round robin fashion. Then, Gooda scales the samples based on the multiplexing factor.

Gooda uses specific events for each supported processor. Only the most interesting and precise events are considered for each processor, which makes Gooda results very precise and complete.

Chapter 3

Performance Counters

This chapter covers the performance monitoring capability of Hardware Performance Counters. It starts with Performance Monitoring Unit (PMU) in general. Then, the Intel® Ivy Bridge microarchitecture is introduced. Finally, the events that are provided by Intel® Ivy Bridge are covered in detail.

3.1 Performance Monitoring Unit

Modern processors contain a PMU. The design and features of each PMU is specific to a given microarchitecture. It exposes two sets of Model Specific Register (MSR). A set to configure the performance monitoring feature and another containing the actual data. Only the first set can be written to. The other set can only be read.

For profiling, the PMU can be used in two ways:

1. Counting: In this mode, all the events that occurred during an application are simply counted. This give a precise information on how well (or bad) an application is behaving. This can also help to compare two programs or two versions of the same software. However, if there is a lot of cache misses for instance, this gives no information about the source of the problem.

2. Sampling: In this mode, whenever an event counter grows higher than a certain number, an interrupt is generated by the PMU. Once the interrupt is caught by the profiler, it can save a sample with the current location in the application and the value of the counters. Modern profilers tend to use the Last Branch

Record (LBR) (see Section 3.1.1) to have access to a precise context, indicating the path to the current location in the program.

To avoid slowing down the processor, the design of modern processor's PMU have been simplified. Often, the instruction associated with an event by the PMU is not the one where the event really occurred. Moreover, the distance between the real instruction and the reported one is variable. Experiments have shown that, even when using advanced PMU features (for instance Precise Event-Based Sampling (PEBS) mode on Intel® processors), events aggregate on some instructions and are missing on others [Chen2010]. Vincent M. Weaver shown that, when the setup is correctly tuned, the values of the performance counters have a very small variation between different runs (0.002 percent on the SPEC benchmarks). Nonetheless, very subtle changes in the setup can result in large variations in the results [Weaver2008].

3.1.1 Last Branch Record

Each Intel® PMU since the Nehalem microarchitecture includes a trace branch buffer, called the LBR. This record is generally implemented as a circular buffer. The LBR captures the source and target addresses of each retired taken branch. It can track 16 pairs of addresses. The LBR also supports filtering to record only the branches occurring at a specified ring level. It can also be filtered by the type of taken branch.

This feature is especially useful when doing sampling. It provides a call tree context for any event. This is a very precise call chain, even with only 16 branches. Using LBR in a profiler results in much more accurate samples.

A complete branch path is also available in the Branch Trace Store (BTS). With this system, all the branches taken are stored in main memory. However, this feature is made for debugging rather than profiling due to its very high overhead, up to forty times slower [Soffa2011].

3.2 Intel® Ivy Bridge

Each microarchitecture has its own performance monitoring features. Depending on the underlying processor, there are plenty of events provided by the PMU. To find the counters that could be interesting for Profile Guided Optimization (PGO), it has been decided to focus on Intel® Ivy Bridge. This is the third Generation

Intel® Core™ processors. At the time of this writing, this is the most recent Intel® microarchitecture.

The Intel® Ivy Bridge microarchitecture is an improvement over the Intel® Sandy Bridge microarchitecture. The microarchitecture is detailed in Figure 3.1.

An assembly instruction that is given to the processor by the branch prediction unit is first decoded and translated to a suite of micro-instruction (uop) by the Micro Instruction Translation Engine (MITE). The instructions are retrieved from the Instruction Cache, which retrieves them from main memory if they are not cached. Then, the processor executes these uops.

A micro-instruction decoded by MITE is cached by the Decoded Stream Buffer (DSB). Once an instruction has been decoded by MITE or recovered from DSB directly, it is put in the Instruction Decode Queue (IDQ). It is faster to get an uop from the DSB than to decode it again using MITE.

Intel® Ivy Bridge supports the latest SIMD (Simple Instruction Multiple Data) operations of SSE and AVX. These instructions perform vector operations on 256 bits at the same time.

An Intel® Ivy Bridge processor has three levels of data cache:

1. L1 cache: 64KB are available: 32KB for data and 32KB for instruction. Each core has its own L1 cache.
2. L2 cache: 256KB unified cache (data and instruction). Each core has its own L2 cache.
3. L3 cache: 3MB to 20MB unified cache. This cache is shared by all cores, the GPU and the system agent.

It also has two levels of TLB cache:

1. L1 TLB: The TLB has support for different sizes of pages (4KB, 2MB and 1GB pages). Depending on the page size, the number of items of the TLB changes. There are one Instruction TLB (ITLB) and one Data TLB (DTLB).
2. L2 TLB: Also called Second Level TLB (STLB). This is a common level for Instruction and Data L1 TLBs.

Intel® Ivy Bridge supports Hyper-Threading. Each physical core is running two virtual core in parallel. Those two virtual cores are sharing the workload on the core.

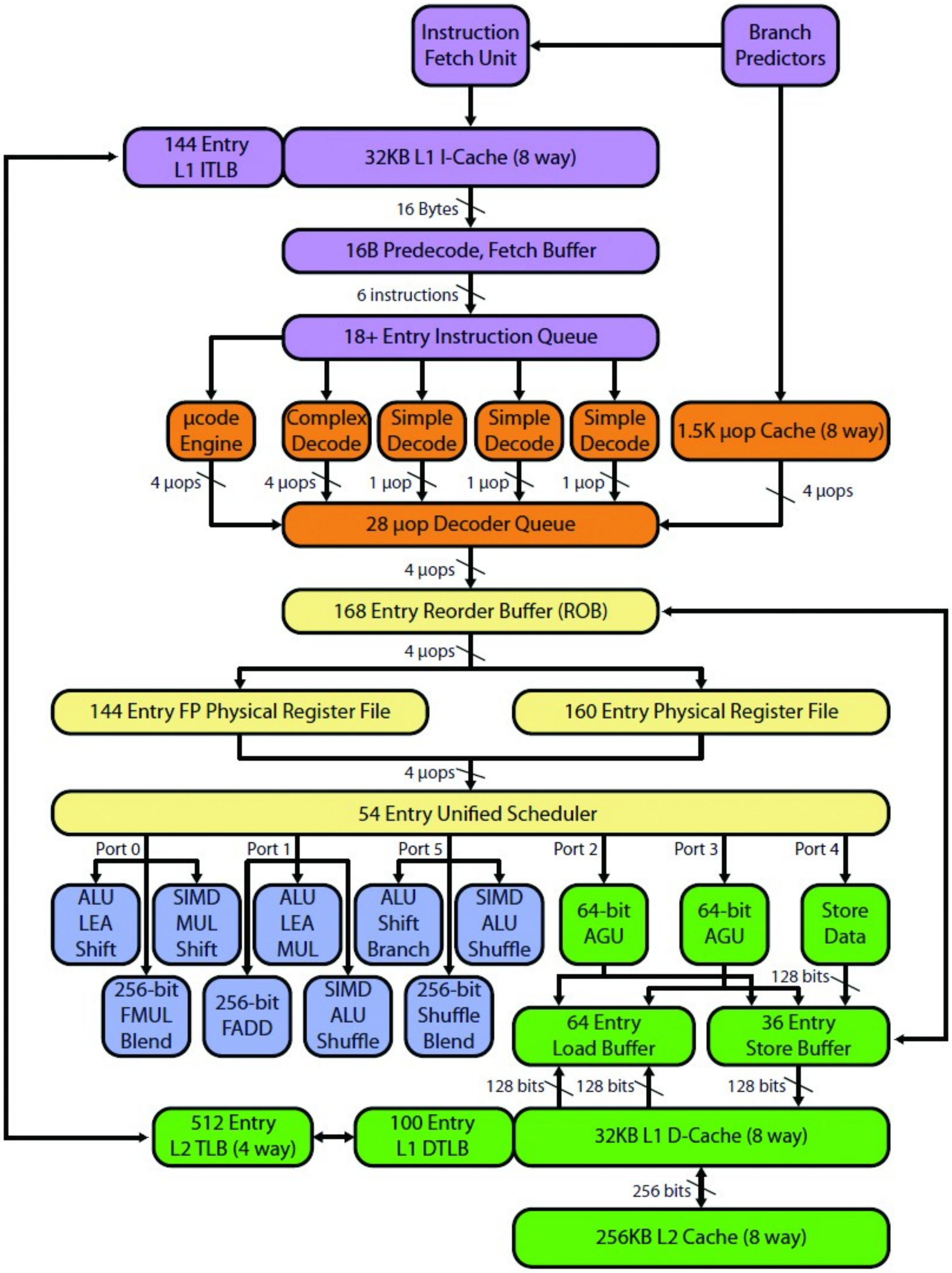


Figure 3.1: Intel® Ivy Bridge microarchitecture

3.3 Events

The complete list of events available on this family of processor is available in Appendix C.

For the sake of clarity, the events are studied here in separate categories.

Some of the events are configurable to be collected by core or by thread. A thread refers here to a hardware thread, activated by Intel® Hyper-Threading technology.

3.3.1 Instructions

The first set of events can be used to gather information about instructions.

The first, and the most used, counter is the number of instructions retired. An instruction is retired when it has been completely executed by the CPU. Some instructions are executed, but their result is not used (for example due to branch misprediction). The number of core cycles as well as the number of unhalted core cycles is also counted.

Some counters are available for SSE and AVX extensions. An important counter is the number of transitions between AVX and SSE (in both directions) that cause performance penalties. Indeed, the hardware must save and restore the upper half of the YMM registers in those cases.

When working with floating point numbers, it often happens that some numbers are difficult to work with (e.g. denormals, NaN, division by zero, etc.). In those cases, microcode are injected to the execution stream. These sequences can be quite long and can be extremely deleterious to performance. When this situation happens, events are issued to count AVX store assists, X87 input and output assists and SIMD input and output assists.

Events specific to branch instructions are covered in Section 3.3.3.

3.3.2 Micro-instructions

Just like instructions, the number of issued uops is also accessible via counters. It is possible to count some specific uops:

- Some instructions that have an increased latency due to flags merge.

- Instructions being slow because of the number of sources (e.g. two sources + immediate), called slow LEA.
- Multiplications and divisions.

There are counters for the number of times an uop is delivered from MITE to IDQ and from DSB to IDQ (the total is available as well). Furthermore, IDQ can receive uops from other sources (e.g. floating point assist). The total of uops delivered to IDQ from any path is counted. It is possible to count the cycles when MITE and DSB are delivered an uop.

Another interesting counter regarding uop decoding is the number of DSB to MITE switches. When control flows out of the region cached in the DSB, the front end switches to MITE to decode instructions. This can have a big penalty.

This processor family has a system to suppress useless move instructions: Move Elimination. The numbers of Integer Instructions and SIMD instruction candidates that were eliminated and that were not eliminated are counted.

The out-of-order scheduler outputs the scheduled micro-instructions to several ports. On this microarchitecture, there are six different ports. Behind these ports are several components that execute uop. It is possible to count the number of uops dispatched to each port. Moreover, the load and store uops can be monitored separately in some of the ports.

3.3.3 Branches

The total number of retired branch instructions is available. Counters with higher granularity also exists. For instance, it is possible to count independently the conditional instructions, the unconditional branches, the calls, the return instructions and the indirect branches.

More important, the number of branch misses is accessible. The processor always tries to predict the branch that will be taken at each conditional branch instruction. This process is done by the Branch Prediction Unit (BPU). The instructions of the predicted branch are already decoded and put into the pipeline before the actual branch is known (only during the execute stage of the pipeline). If the prediction is incorrect, all the instructions following the conditional branch have to be canceled and the pipeline starts over with the correct instructions. As modern processors have quite long pipeline (14-stages pipeline for Intel® Ivy Bridge), it incurs a big delay. Just like branch counters, it is also possible to count separately the misses for different types of branches.

3.3.4 Caches

A lot of events are related to caches and memory.

The DSB is sometimes referred as a L0 cache. However, the related events are treated in Section 3.3.2.

For each level of cache, the number of misses is recorded. The source of the miss is also sometimes available in a specific counter (e.g. Request For Ownership (RFO) missing L2). The number of hits is recorded. There are specific counters to record where does the hit come from (e.g. Instruction Fetch hitting L2 cache).

Moreover, there are several of counters recording specific events:

- Writebacks from a cache to the upper cache
- Partial misses. For instance, L1 miss hitting L2 cache.
- The cycles in which a Data Cache is locked.
- Read For Ownership (RFO) requests.
- Hardware Prefetch requests. The processor contains a prefetcher that tries to detect which data will be accessed in the future and prefetch these data into one level of cache. Events are recording those prefetch requests. Counters are also recording the loads accessing the prefetched data.
- Retired load uops. Several events are managing retired load uops with specific data sources (e.g. Retired load uops hiting Last Level Cache (LLC)).
- Line eviction. Events are keeping track of dirty and clean L2 lines evicted by demand or by the prefetcher.
- Stalls. When an instruction needs to wait for a data to be computed or to be fetched from main memory, there is a so-called stall. A number of events are recording stalls occurring for several reasons (e.g. the Instruction Queue (IQ) is full, the prefix length of the instruction has to be changed, the Re-Ordering Buffer (ROB) is full, etc.).

When unaligned data is accessed, it can happen that the information is crossing two cache lines, this is called a cache line split. This can cause an instruction to run two to four times slower. Some events are recording these accesses.

There are events recording the number of DTLB misses. The events are separated between load and store operations. Moreover, for both operations, two kinds of events are recorded:

1. DTLB misses hitting STLB
2. DTLB misses missing STLB: miss at all TLB levels, causing a page walk

For ITLB, there are only events monitoring the number of ITLB misses at all levels.

For both TLB caches, there are events when the cache is flushed. This usually happens when the operating system operates a process switch.

3.3.5 Others

The processor implements a security level, called the Current Privilege Level (CPL). The value of the CPL is called a ring with a value from 0 to 3. It is possible to know the number of clock cycles when the thread is in ring 0 and when it is in another state (1,2,3).

For each possible thread (8 for this processor), the numbers of cycles during when the thread is not halted is recorded.

The microarchitecture separates the cores, the GPU, the system agent and the LLC. The system agent, also called uncore, is responsible for the communication with the Power Control Unit, the PCIe devices, the display, etc. The cores can communicate to the system agent. When that happens, events are recording the type of communication that is made.

When a program is modifying itself, it writes to a code section. Then, the entire pipeline has to be cleared, as well as the trace cache. This incurs a big performance penalty. An event is thrown when a program is modifying itself.

3.4 Useful events

From the whole list of events, some events have been selected as especially useful for a compiler. The following list shows the events that can cause enough performance penalties to try to avoid them automatically.

- DSB to MITE switches: If it happens often, this can indicate a hot region of code not fitting in DSB. However, it is hard to find exactly why this event happens. It can come from the front end, or it can come from a loop body with too many instructions or from bad branch predictions.
- Transitions between AVX and SSE. This can indicate a bad mix of SSE/AVX in the code. Nevertheless, this can generally be avoided at compile-time without feedback.
- The number of Floating Points assists. Nevertheless, they are generally caused by denormals and can be avoided using the Flush To Zero feature of the processor.
- Instruction Cache misses. A cache miss implies a long delay in the pipeline. Nevertheless, the ifetch event is not precise. Indeed, the event is often thrown hundred of instructions later, so it is almost impossible to find the faulty instruction.
- The different cache (L1, L2, LLC, TLB) misses. A LLC miss can have a big impact in the performance of an instruction.
- Events indicating unaligned data. Unless the padding in data structures is specified by the programmer itself, this case should not happen. Indeed, compilers already rearrange fields in order to avoid having unaligned fields.

Given these events and their drawbacks, the project will focus on data cache-misses, and especially the LLC misses, which are the most deleterious to performance and the most practical to work with.

Ideally, the implementation should make it possible to add support for more events in the future without too much effort.

Chapter 4

GCC

This chapter presents the detailed analysis of the GNU Compiler Collection (GCC) compiler.

All the information of this section is based on GCC 4.7.2 that was the last stable version at the beginning of this analysis.

4.1 Architecture

GCC architecture is composed of three parts:

1. Multiple front ends. GCC supports compilation of several programming languages. Each language has its own front end. The main distribution contains front ends for C, C++, Objective C, Fortran, Java, Ada, and Go.

2. A common middle end. It is common to all front ends and back ends. It provides most of the optimization capabilities of GCC.

3. Multiple back ends. GCC supports many targets. Each target is implemented as its own back end.

The biggest advantage of this architecture is that, to add support for a new programming language, it is only necessary to add a front end. A front end can use any back end and so, generate code for any supported platform. Most of the optimization passes are done in a language-independent way, so that every front end takes advantage from them.

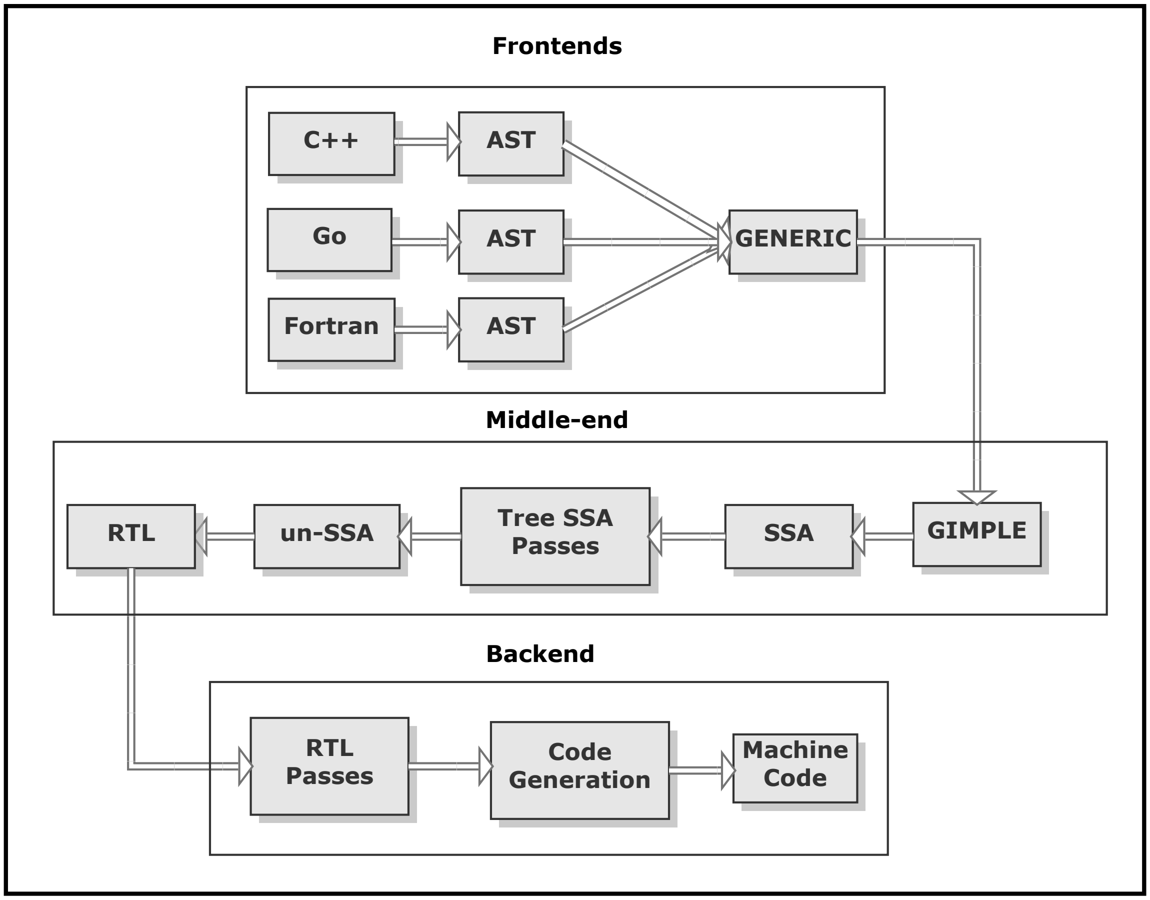


Figure 4.1: GCC Architecture

Figure 4.1 outlines the main components of the GCC architecture.

The front end parses the entire output to an intermediate representation. Each front end is free to use any intermediate representation, but the C and C++ front ends use the same representation, GENERIC.

Once the front end has finished the parsing, semantical analysis and possibly some language-dependent passes, it passes the intermediate representation to the optimizer. The optimizer uses a specific Intermediate Representation (IR), GIMPLE. The front end representation has to be gimplified (official term for a conversion to GIMPLE) before being passed to the optimizer. Then, several transformation passes are run on this representation to produce a program as fast or as small as possible. The middle end is also responsible of handling high-level languages like OpenMP or the Transactional Memory extension.

When all the passes of the optimizer have been performed, the program is passed to the back end. Again, GCC has a specific intermediate representation for this part, called Register Transfer Language (RTL). The GIMPLE representation is transformed into RTL. Some other low-level optimization passes are applied to the RTL program. Finally, once all the transformations have been performed, the machine code for the function is written to the object files. These two parts (optimizations and code generation) are not directly separated, but are implemented as passes in the Pass Manager.

4.2 Intermediate Representations

GCC uses three major IR: GENERIC, GIMPLE and RTL.

The first one, GENERIC, is a language-independent structure used to represent an entire function into a tree. It is the responsibility of the front end to generate a GENERIC tree from the source file. However, this representation is quite complex and some of the passes would be too hard to implement on it. This is why the middle end uses another representation. Not all front ends use GENERIC.

GIMPLE is the intermediate representation used by the middle end. It is also a tree structure, derived from GENERIC, but with some several structural restrictions. This is basically a Three Address Code (TAC) representation. No expression can contain more than three operands and function call parameters can be only variables or constants, etc. This representation is derived from the SIMPLE IL used by the McCAT compiler [Hendren92].

Before the optimization passes, the GIMPLE tree is transformed into Static Single Assignment (Static Single Assignment (SSA)) form. GCC implements the SSA form as described in [Cytron1991]. It is not another representation, it is still a GIMPLE representation, but with more restrictions that make all statements SSA. In this form, each variable is assigned exactly once. It also introduces PHI nodes representing the set of values that a variable can hold. This form makes some optimizations easier to implement and generally more efficient as well. Moreover, in SSA form, UD Chains are explicit.

During Loop Optimizations, the SSA form is updated to a Loop Closed SSA Form (LCSSA). LCSSA is an SSA form with an extra condition: No SSA name is used outside of the loop in which it is defined. This representation has several benefits specific to loop optimizations:

- The SSA names are directly linked to the loop they are declared in, making Induction Variable analysis easier.
- The new PHI nodes make it easy to find the values defined in the loop but used outside of it.
- Unless new values need to be created inside the loop, updates of the SSA form can be made locally.

The last IR is RTL. This is a low-level representation. It corresponds to an abstract target architecture, close to the hardware. This representation can be seen as an abstract assembly language represented in algebraic form.

4.3 Optimization

The program is optimized in two phases. First, in its GIMPLE representation and then in its RTL representation. Some language-dependent optimization are also made on the GENERIC representation.

Each optimization is described into a pass. Each pass defines what it needs, what it provides, when it should run and what it does. The pass manager is responsible for running the adequate passes in the correct order. There are two main types of passes:

- **Tree SSA passes**: These passes are run in the GIMPLE representation. The first passes are lowering some constructs, building the appropriate data structures and look for warnings in the code. Then, the GIMPLE tree is rewritten in SSA form. After that, the optimization passes start.
- **RTL passes**: These passes run after the Tree SSA passes. The first pass performs GIMPLE to RTL transformation. Then, there are the optimization passes. Finally, the last passes are generating the assembly code for the target platform.

The Control Flow Graph (CFG) is always kept up to date by the passes. The CFG is only built once.

Section 4.3.1 and Section 4.3.2 give detailed information about Tree SSA passes and RTL passes, respectively.

4.3.1 Tree SSA Optimization Passes

This section presents all the optimization passes performed on the GIMPLE representation of the program, in SSA form.

They are applied in a specific order:

1. Small Interprocedural optimizations are applied to the whole program.
2. Regular Interprocedural optimizations are applied to the whole program.
3. Late Interprocedural analysis passes are applied to the whole program.
4. Intraprocedural optimizations are performed on each function output to assembly.

Some passes are run several times throughout the optimization process. There are also different versions of some optimizations. For instance, Constant Propagation is made once at Interprocedural level, once at Intraprocedural level and some times in a form taking conditional branches into account.

The following list presents the optimization passes performed by GCC. This list does not contain the utility passes (analysis pass, cleanup pass or transformation for later passes) neither the debug and warning passes. Moreover, this list contains only the first instance of an optimization technique.

A complete list (but not detailed), is available in Appendix D.

- Inlining. It is made in two versions, one as the first optimization (the early inliner) and one later with whole program knowledge (the real inliner).
- Conditional constant propagation. Propagate constants with handling of conditional branches.
- Forward propagation of single-use variables. It is basically a specialized form of tree combination. It replaces some series of instructions by a smaller series of instruction combining them.
- Scalar replacement of aggregates. Replace some local aggregates into a set of scalar variables. This optimization is made to improve the efficiency of following optimizations. This pass is run in several flavors: Early Intraprocedural, Interprocedural and again Intraprocedural.

- Full Redundancy Elimination. Eliminate computations redundant on all paths.
- Conditional copy propagation. Propagate copies with handling of conditional branches.
- Merge PHI Nodes.
- Dead Code Elimination. Remove statements without side effects whose result is unused.
- Tail recursion elimination. Replace tail recursion with loops when applicable.
- Switch conversion. If a switch contains only constant assignment to variables, it is possible to replace it with an array of these values and then get the value with the index of the case.
- Profiling pass. Read the profile if in PGO mode, otherwise predict it.
- Split Functions. Split function bodies to improve inlining. If the function is made of two parts: one big, one small, conditionally executed, the pass can move the big part in a new function. The source function can now be inlined with greater benefits when the small part is executed.
- Tree Profiling pass. Insert instrumentation instructions into the program.
- Matrix flattening. Transform multi-dimensional arrays into flat arrays.
- Merge multiple constructors/destructors.
- Complete unrolling of some inner loops.
- Conditional dead call elimination. Some builtin functions are setting `errno` on error conditions and therefore are not pure. Even if dead, these calls cannot be eliminated. However, if the compiler knows what are the conditions causing this effect, it can replace the calls by a conditional branch on this condition and thus the call is only executed if there will be an error. This is done when the return value of the function is not used.
- Return slot optimization. Avoid memory copy by passing the address of an object to the function instead of returning this object.
- PHI node propagation. Propagate indirect loads through the PHI nodes.

- Value Range Propagation. Propagate the ranges a variable may hold and perform optimization based on these ranges.
- Transform conditional stores in unconditional ones.
- Combine if expressions. Combine several conditional branches into one. This simplifies the control flow of the program.
- PHI optimizations. Try to use conditional expressions by recognizing some forms of PHI inputs.
- Loop Header Copy. Copy the header of a loop, which eliminates a jump and provide opportunities for better Loop Invariant Motion.
- Optimization of stdarg functions. Optimize standard functions having variable number of arguments in parameters by saving as less registers as possible.
- Lower complex arithmetic. Rewrite complex arithmetic operations into simpler ones.
- Dominator Optimizations. Perform trivial optimization based on dominators: copy/constant propagation, jump threading and redundancy elimination.
- Eliminate degenerated PHI. After some transformations, some PHI nodes are degenerated (they contain only one variable or constant) and therefore can be rewritten as a simple assignment.
- Dead store elimination. Eliminate stores to memory that are not used and overwritten.
- Reassociation. Rewrite arithmetic expressions to enable other optimizations.
- Optimize builtin object size computations.
- String length optimizations.
- Optimization for sin/cos operations.
- Partial Redundancy Elimination. Eliminate partially redundant computations.
- Code sinking. Stores and assignments are moved closer to their use point. This usually helps register allocation.

- Loop optimization. This pass is a complex pass performing several optimizations.
    - Loop Invariant Code Motion.
    - Dead Code Elimination in loops.
    - Loop unswitching. Move invariant conditional jumps out of the loop.
    - Constant propagation using Scalar Evolution. Transform dependent induction variables to simpler version.
    - Loop distribution. Break a big loop into multiple smaller loops.
    - Graphite Optimizations. Graphite is a framework for loop transformation, based on the polyhedral model.
        - Graphite linear transformations. Perform several transformations of loops using polyhedral model.
    - Canonical induction variable creation. Create a simple counter variable for the number of iterations of the loop.
    - Vectorization. Transform loops to operate on vector types instead of scalar types.
    - Predictive commoning. Reuse computations from previous iterations of the loop.
    - Complete unrolling.
    - SLP Vectorization. Perform basic block vectorization.
    - Autoparallelization. Split the loop iteration to run into several threads. The loops are parallelized using OpenMP.
    - Array prefetching. Add prefetch instructions to loops to improve cache efficiency.
    - Induction variable optimizations. Perform strength reduction, merging and elimination on the induction variables.
- Tail duplication for superblock scheduling. Duplicate common parts of code among basic blocks to create a so-called super block. A super block is a block with one entry point and multiple exit points. It can make instruction scheduling much more effective for large pipeline.
- Fold built-in functions.

- Optimize widening multiplications.
- Tail call elimination. Replace tail call with a jump when applicable.
- Unpropagate edge equivalences. Perform transformations to avoid redundant statements after the program has left SSA form.
- Return value optimization. Avoid copy when returning an object from a function.

Some of the passes require target support and so, delay the actual transformation to the RTL passes. In those cases, the passes set some flags to the GIMPLE tuples to let RTL passes optimize them.

4.3.2 RTL passes

This section presents the optimizations performed on the RTL representation of the program.

The optimization passes are separated in two phases:

- A set of optimizations is made at the beginning just after the RTL representation is generated.
- Another set of optimizations after Register Allocation is finished.

Again, this list contains only the optimization passes, all the other passes are ignored.

- Control Flow Cleanup. This pass performs several small optimizations: remove unreachable code, simplifies jumps, merges basic blocks, etc.
- Decompose multi-word registers.
- Common Subexpression Elimination. Remove redundant computations within basic blocks. It is done once locally and once globally.
- Forward propagation of single-def values.
- Partial Redundancy Elimination.

- Code Hoisting. Eliminate expressions computed in multiple code paths from a single point.
- Global Copy/Constant Propagation.
- Store Motion. Move stores close to their usages.
- If conversion. Replace some conditional branches with non-branching equivalent code when applicable.
- Loop analysis pass
  - Loop Invariant Code Motion.
  - Loop Unswitching. Move an invariant conditional branch out of the loop.
  - Loop Unrolling.
  - Loop Peeling. Perform some iterations of the loop out of the loop. This pass facilitates other optimization.
  - Loop instructions optimizations with low-overhead instructions
- Forward Propagation of addresses.
- UD Chain based Dead Code Elimination.
- Partition blocks in hot/cold sections.
- Split instructions.
- Mode switching optimization. Minimize the number of mode changes required.
- Swing Modulo scheduling. Perform transformations to improve pipelining in loops.
- Redundant Extension Elimination.
- Compare Elimination. Remove comparisons when the flags are already available for this comparison. For instance, some arithmetic instructions are setting the flags necessary for a comparison.
- Branch Target Register Load Optimization. Hoist loads out of loops and enable interblock scheduling.

- Combine Stack Adjustments. Reduce the number of stack adjustments by propagating them directly to the addresses.
- Peephole Optimizations.
- Register Renaming. Rename registers to avoid register pressure.
- Copy Propagation on hard registers.
- Fast Dead Code Elimination.
- Basic block reordering. Reorder basic blocks to improve Instruction Cache efficiency.
- Leaf Regs Analysis. Determine if a function is a leaf function and if it uses only leaf registers. This is used during code generation.
- Instruction scheduling. Perform transformation to avoid pipeline stalls.
- Delayed branch scheduling. Fill the delay slots of some instructions of RISC processors.

4.4 Profile-Guided Optimization

GCC is able to use two kinds of profiling:

1. **Edge profiling**: Collect the frequencies of the CFG edges.
2. **Value profiling**: Collect the possible values of some variables. For each possible value, the frequency of its usage is also recorded.

Both kinds of profiling can be performed at the same time.

This profiling is done by inserting code into the program to collect information.

Once the profile has been generated, it has to be given to GCC for the next compilation pass. The profile information is stored inside each `basic_block` (the GCC structure representing a basic block). Each structure has two integer fields:

- `frequency` Estimation of how often the basic block is executed within a function. The frequency is scaled from 0 to `BB_FREQ_BASE`. During optimization, the frequencies can change (loop unrolling or cross-jumping optimization can cause this behavior).
- `count` The number of executions during the training run. This number is zero if there is no available profile.

Each edge of the CFG contains also a branch probability field. This is the probability that the control will flow from the source basic block to the target basic block.

The value profile is stored into hash tables. Each profile value is stored into sets of histograms stored themselves in the hash tables. There is one hash table per instrumented function.

When the profile is not available, the compiler attempts to predict the behavior of the program by using a set of heuristics. The estimated frequencies of each basic block are propagated over the control flow graph until every probability is known. The hard count of the basic blocks is not computed.

Optimization passes do not perform differently if there is a profile or not. If no profile is available, the profile is filled by the prediction system and the optimization passes will use this static profile.

Several passes are using the profile information:

- **Register Allocation** GCC uses a simple priority-driven register allocator working in two passes. The basic block frequencies are taken into account to compute the priority of each pseudo register and spill code in the least frequently used basic blocks.
- **Basic Block Partitioning** Each basic block is assigned a state between “maybe hot”, “probably cold” and “probably never executed”. This state is used to avoid optimizations trading code size for performance in cold code.
- **Function Reordering** Based on its basic blocks, the function is put in separate sections of the ELF file (hot together and cold together).
- **Basic Block Reordering** This pass puts basic blocks that are part of an often used path, close in the generated object file.
- **Loop Unrolling** This pass avoids to unroll cold loops. The unroll factor can change depending on the hotness of the loop.

- **Loop Peeling** This pass avoid to peel cold loops.
- **Inlining** The priorities for inlining are computed using the frequencies of the call instructions.
- **Tail Duplication** This pass uses the basic block frequencies to compute high frequency paths and apply Tail Duplication in it.
- **Operation Specialization** When a value is known to often be a constant, it is possible to optimize some operations. For instance, if there is a division by b and b is known to be frequently one, it is possible to add a branch to test if it is equal to one and simplify the division to an assignment.
- **Indirect/Virtual Call Specialization** When an indirect function call leads most of the time to the same function, it is possible to add a conditional branch testing the type and call directly the function if it is the right type. The direct call can now be inlined if it makes an interesting candidate.

With all these optimizations put together, Profile Guided Optimization (PGO) with an accurate profile can bring a good improvement to intensive applications.

Chapter 5

Memory Optimization

It has been decided to focus on memory optimizations by the use of the cache misses information coming from the Hardware Performance Counters.

A compiler has several means to improve usage of memory hierarchies [Srikant2007]:

1. Software Prefetching [Callahan1991]: Add prefetch instructions to the code to make sure that the data is in the cache when it is accessed. This avoids stalls in the pipeline.

2. Loops Transformations [McKinley1996]: Transform loops working on arrays to make the access pattern more efficient and therefore limit the number of cache misses. This improves the locality of the memory references. Some of these transformations can also improve the efficiency of other transformations and enhance the vectorization and parallelization capabilities of the loops. Generally, it makes the loops iterate with the memory layout.

3. Data Layout Optimization [Rubin2002]: Change the layout of structures, by reordering the fields or by splitting/peeling structures in several smaller structures. The transformations are done to improve the locality of the fields.

4. Code Layout Optimization: Change the order of basic blocks and functions to have as few Instruction Cache misses as possible.

All these techniques aim at reducing the number of cache misses, for both Instruction and Data caches.

The Code Layout optimizations of GCC already use profile to improve their efficiency. Possibly, the number of Instruction Cache misses could backup the decision of Basic Block Reordering and Function Reordering. However, no case has been found where the Instruction Cache Misses can do a better job than the frequency of the edges. GCC does not perform full ordering of basic blocks and functions, it only separates them in hot and cold sections in the generated executable.

GCC does not perform any binary layout transformations at the time of this writing, but there have been attempts to implement it [Hagog2005, Golovanevsky2007]. Nothing is available in GCC trunk, the efforts have been made in the `struct-reorg` branch of GCC. Nonetheless, this branch has not been updated for several years. It is a well-known optimization and it has been covered widely in research [Rubin2002, Hundt2006]. Nevertheless, it requires a complete view of the program (Whole Program Optimization or Link Time Optimization) to make the changes, and so is not easy to implement and not always convenient for the user. For these reasons, this optimization has not been covered in detail.

From this analysis, it has been decided to study in detail loop transformations and how to use cache misses in one of the optimization techniques.

5.1 Loop Fusion

Loop Fusion, sometimes called Loop Jamming, is a loop transformation merging several loops into one.

This optimization is not commonly supported by Compilers. However, it has been well researched [Carr1994, Singhai1996, Marchal2004, Fraboulet2001].

Loop Fusion has several advantages. The most important is that, if both loops are working on the same array, it generally results in an improved temporal locality and thus, less cache misses. Depending on the loops, it can result in both spatial and temporal locality and also better reuse. It also decreases the overhead of loop instructions as the loop is done only once instead of several times. Moreover, it can also bring new opportunities of optimization in the loop body, for example for Common Subexpression Elimination. A bigger block of code can bring more opportunity for instruction scheduling. The generated code is generally smaller than the sum of the original bodies, but that depends on further optimizations. Loop Fusion is especially well suited to work with Loop Permutation [Carr1994]. It can create perfect loop nest, enabling a Loop Permutation with better spatial locality. The combination

```
for $i = 1 \rightarrow SIZE$ do
    $A(i) \leftarrow i$
end for
for $i = 1 \rightarrow SIZE$ do
    $B(i) \leftarrow A(i)$
end for
```

(a) Before fusion

```
for $i = 1 \rightarrow SIZE$ do
    $A(i) \leftarrow i$
    $B(i) \leftarrow i$
end for
```

(b) After fusion

Figure 5.1: Loop Fusion Example

of the two transformations may result in a loop with better spatial and temporal locality.

Loop Fusion has also some drawbacks. The first is that the code in the loop can become quite large and cause more Instruction Cache misses. Moreover, on some architecture, it is faster to initialize two arrays in two different loops than in one due to improved cache locality.

Not all loops can be merged together, there are several conditions to be met before two loops can be fused:

- The loops have to be at the same nest level.
- The iteration pattern needs to be the same.
- No dependencies are violated by the merge.
- The loops are always executed in pair. Once the first loop is executed, the second loop is always executed and the second loop is never executed alone.

Even if it is not always mandatory, Loop Fusion generally merges a couple of parallel loops or a couple of sequential loops together.

Figure 5.1a shows an example of code profiting from this optimization, A and B being arrays of size $SIZE$. Once Loop Fusion has been applied and Constant Propagation has been performed on the resulting code, the program will resemble the code shown in Figure 5.1b.

The resulting program should normally be much faster than the first one. There will be twice less accesses to $A(i)$ and loop instructions (increment, comparison and jump). The first point being the more interesting because it can importantly reduce the number of cache misses. Indeed, cache misses are much more detrimental to performance than loop instructions.

```
for $i1 = 1 \to 4$ do
    for $i2 = 1 \to 4$ do
        $A(i1, i2) \leftarrow A(i1, i2) + 5$
    end for
end for
```

(a) Before Loop Interchange

```
for $i2 = 1 \to 4$ do {Loop Swapped}
    for $i1 = 1 \to 4$ do
        $A(i1, i2) \leftarrow A(i1, i2) + 5$
    end for
end for
```

(b) After Loop Interchange

Figure 5.2: Loop Interchange example

5.2 Loop Interchange

Loop Interchange (or Loop Permutation) swaps the order of two nested loops. The first purpose is that the outer loop carries the data dependencies so that the inner loop is parallelizable. Another purpose is to improve the memory layout access to arrays to reduce cache misses. It generally improves the spatial locality of the array references.

For example, the program in Figure 5.2a does a poor usage of the cache by iterating against the memory layout. This is only true in the case of column major multidimensional array, like in Fortran. It can be optimized using Loop Interchange, resulting in the program shown in Figure 5.2b. The resulting program iterates with the memory layout, resulting in a much better spatial locality.

5.3 Loop Fission

Loop Fission (or Loop Distribution) is a transformation doing the opposite of Loop Fusion. It transforms a single loop into several smaller loops. Loop Fission does generally not bring a lot of performance benefits in itself, but may enable other transformations. It can reduce the number of Instruction Cache misses by transforming a big loop into several smaller ones. In the same case, it is also possible to reduce register pressure. If the loop is not parallel, it is possible to create some parallel sub loops. It may also bring new opportunities for vectorization on the small loops. It may also enable Loop Permutation on some of the generated loops.

When a loop works on several arrays and when the instructions are not depending on each other, it can very profitable to distribute the loop. The resulting loop would have a better cache behavior.

```
for $i1 = 1 \to m$ do
    for $i2 = 1 \to m$ do
        for $i3 = 1 \to m$ do
            $R(i3, i1) \stackrel{\pm}{=} A(i2, i1)B(i3, i2)$
        end for
    end for
end for
```

(a) Before Loop Tiling

```
for $k = 1 \to m$ by $b$ do
    for $j = 1 \to m$ by $b$ do
        for $i1 = 1 \to m$ do
            for $i2 = k \to min(k + b - 1, m)$ do
                for $i3 = j \to min(j + b - 1, m)$ do
                    $R(j, i1) \stackrel{\pm}{=} A(k, i1)B(j, k)$
                end for
            end for
        end for
    end for
end for
```

(b) After Loop Tiling

Figure 5.3: Loop Tiling for Matrix Multiplication

5.4 Loop Reversal

Loop Reversal changes the order in which a loop iterates. The main advantage is that it may enable other optimization to perform due to the changed dependencies. This can also be useful on some hardware containing instructions to decrement a register and jump if its value is not zero, creating a loop in one instruction. Generally, it does not improve data locality.

5.5 Loop Tiling

Loop Tiling (also called Loop Blocking) partitions the iteration space of a loop into smaller blocks fitting in the cache. This can improve both spatial and temporal locality of array references.

The best example for this optimization is the case of Matrix Multiplication. Figure 5.3a shows a Square Matrix Multiplication. When the matrices are large, this program performs poorly because too much memory is shuffled around. Figure 5.3b shows the version optimized by Loop Tiling. As the blocks are used several times, the temporal locality is improved. The spatial locality is also improved because elements with consecutive addresses tend to be used together.

```
for $i = 1 \rightarrow N$ do
    for $j = 1 \rightarrow N$ do
        $A(i, j) \leftarrow A(i-1, j) + A(i, j-1)$
    end for
end for
```

(a) Before Loop Skewing

```
for $i = 1 \rightarrow N$ do
    for $j = i+1 \rightarrow i+N$ do
        $A(i, j-i) \leftarrow A(i-1, j-i) + A(i, j-i-1)$
    end for
end for
```

(b) After Loop Skewing

Figure 5.4: Loop Skewing Example

5.6 Loop Skewing

Loop Skewing works on a couple of nested loops. The outer loop iterates i times and the inner loop iterates j times. Loop Skewing by a factor f adds $f * i$ to the upper and lower bounds of the inner loop. It subtracts $f * i$ from all accesses to j. It does not improve the memory access layout. This technique changes the memory layout from square to rhomboid.

The dependency direction vectors are changed and this may enable other loop optimizations, and especially, this may make the inner loop parallelizable. This optimization is complex and rarely implemented in compilers.

For instance, the program in Figure 5.4a, can be optimized using Loop Skewing by a factor 1, resulting in the program in Figure 5.4b. In the original version, the direction vectors are $(=, <)$ and $(<, =)$, making Loop Interchange impossible. Once the loop is skewed, the direction vectors become $(=, <)$ and $(<, <)$ making Loop Interchange legal.

Chapter 6

GCC Sampling PGO

This chapter presents the implementation that has been performed to make it possible to use Sampling-based Profile Guided Optimization (PGO) in GCC. Moreover, this implementation also adds support for cache misses into GCC PGO profile. This implementation uses Hardware Performance Counters to profile the application and generate the profile for GCC.

At the time of this writing, GCC does not support sampling-based PGO. The implementation presented in this chapter is based on two other projects:

1. Generic Optimization Data Analyzer (Gooda) (See Section 2.7). Gooda analyzes the results of profiling an application with `perf` and produces a set of spreadsheets describing the behavior of the application. The data generated by Gooda is more adapted than the data provided directly by `perf`.

2. AutoFDO patch[1] (AFDO). This patch has been implemented by Google to add sampling-based PGO to GCC. The input is a binary file in a specific format, containing all the profile information. Unfortunately, Google uses an internal tool to convert `perf` samples to AFDO format. Google has not yet released any application to generate this input file. The Google implementation is based on the `perf` profiler and uses debug information to recreate the complete profile from the binary. This project proposes an implementation, presented in this chapter, to replace this tool. The AFDO patch has been committed to the Google branch of GCC, but is not yet available in the trunk of GCC. AFDO is similar to the SampleFDO technique presented in [Chen2010], but do not use

[1] `http://gcc.gnu.org/wiki/AutoFDO`

the Minimum Control Flow technique.

With this implementation, PGO is made in several steps:

1. The application is launched with `perf` to collect the profile. `perf` should not be launched directly, but should be launched using one of the Gooda collection scripts. These scripts already contain the necessary options for `perf`.
2. Gooda is used to generate spreadsheets from the `perf` data (See Section 6.1).
3. The spreadsheets are then converted in a file in the AFDO format (See Section 6.2). The conversion is made using an application written in C++, the gooda-to-afdo-converter (See Section 6.3).
4. GCC reads the profile with the AFDO patch and optimizes the code again using the new knowledge.

This implementation has been done in two steps. First, only the information about frequencies have been considered. This step does not need any change in GCC. Finally, the information about cache misses have been included into AFDO and GCC. Two steps have been made to make sure that the results obtained at the first step were consistent with the results observed by Google during their testing of AFDO, before modifying the profiles with cache misses support.

The version of GCC used for all the experiments presented in this chapter is the GCC 4.7 Google branch[2].

6.1 Gooda Spreadsheets

Gooda generates several so-called "spreadsheets". A spreadsheet is a text file containing data for a specific view. Each spreadsheet is encoded in a pseudo JavaScript Object Notation (JSON) format. A file represents a matrix (or array of array) of data. An array starts with a [character and finishes with the] character. Each value of the array is separated by a comma. In each file, the first rows are headers containing information related to the analysis, for instance the periods of sampling or the penalties of each event. After the headers, the values are counter values. The content of each row of the array can be viewed as a line of a Comma Separated Values (CSV) file.

[2] `http://gcc.gnu.org/svn/gcc/branches/google/gcc-4_7/`

Gooda generates different spreadsheets:

- function_hotspots.csv: Indicates the functions that are the most interesting candidates to optimize. A summary of all the counters is given for each function.
- process.csv: Summary of information for each of the processes running during the record. Again, the counter values are summarized for each process.
- asm/x.csv: The assembly view spreadsheet for the xth hotspot function. In this file, each assembly instruction is associated with the values of the counters.
- src/x.csv: The source view spreadsheet for the xth hotspot function. This view is similar to the assembly view except that it displays source instructions.

Only hotspot functions are represented in the spreadsheets. All the other functions are skipped. This is not a problem in this case because they are the important functions to optimize and so the ones that could profit the most from Profile-Guided Optimization. By default, the 20 most costly functions are taken as hotspot functions. Moreover, when Gooda detects that there are many functions, this limit is increased to 200. If necessary, this can also be configured when running Gooda.

6.2 AutoFDO Input file

The format of AFDO is based on the GCOV format, used for the traditional profile files. It is not the same format, but it uses the same data structure and the same encoding convention. This format is partly documented in the `gcov-io.h` file of the GCC source tree.

The description of the format has been obtained by reverse engineering the patch.

This format uses three data types:

- `unsigned`: Encoded in 32-bits binary form using the endianness of the generating machine.
- `counter`: 64-bits number stored as two 32-bits numbers.
- `string`: Encoded in NULL-terminated ASCII format.

The file starts with a header containing three fields:

- magic (unsigned). Indicates the type of file. Must be `GCOV_DATA_MAGIC`.
- version (unsigned). Indicates the version of GCOV. Must be the same version of GCOV that GCC is using.
- stamp (unsigned). The stamp is used by GCOV to synchronize the data and note files. It is not used by AFDO.

After the header, the data of the profile is separated into different sections. Each section starts with a tag indicating the type of the section, followed by the length of the section and then its data. The length of the section is skipped by AFDO, but is necessary to make it a valid GCOV file.

An AFDO file is made of four sections:

1. The file name table. Although named file name table, this section stores strings. This section contains a list of all the file and function names used by the later sections. When a string is used in another section, only an index is used. This index refers to the position of the string in the file name table.

2. The function profile. This section contains all the functions. For each function, the file it is located in and its name are saved. The number of time this function has been executed as well as the total count of the basic blocks of the function are also available. Each function has several so-called "inline stacks". An inline stack is a kind of call stack. If the source instruction is a call that has been inlined, the inline stack will contain source locations from both the callee and the caller function. The profile maps each inline stack to its count. A source location is defined by a file name, a function name, a line number and a discriminator. For each inline stack, the number of dynamic instructions it is made of is also available. A dynamic instruction is an assembly instruction generated by the compiler. Most of the time, several dynamic instructions are generated for a single line of source code.

3. The module information. This section contains the list of all the source files of the application and several fields for each of them (e.g. the includes, the definition, etc.). For the time being, this information is not used in GCC and so will not be filled by the implementation (the number of modules is always set to zero). Google plans to use this information in 2013.

4. The working set. This section contains 128 pairs of counters. These counters indicate how many profiled counts compose the majority of the program's execution count. It indicates how large is the dynamic hot code region during runtime. This information is used to guide code size based decisions (e.g. unroll and peel), to prevent them from increasing the code size too much. This can be viewed as a map from *bucket* to the total number of instructions consuming *bucket*/128 of dynamic instructions. The working set is generated from the hottest instructions of the program. For instance, if the bucket 127 is 55, 55 instructions account for 127/128 ← 99.21% of the dynamic instructions of the program.

The most important part of the profile is the function profile, it has to be filled very accurately in order for AutoFDO to identifies correctly the instructions.

Figure 6.1 presents an example of source code that may be encountered by the toolchain.

```
void a() { //Executed 10 times
    bar();
    foo();
}

void foo() {
    int total = 100;
    int i = 0;

    while (1) {
        if (i++ < total){
            break;
        }
    }
}
```

Figure 6.1: Source code example for AFDO profiling

```
t.c:2  --> 10
t.c:3, t.c:7   --> 10
t.c:3, t.c:8   --> 10
t.c:3, t.c:10  --> 1000
t.c:3, t.c:11  --> 1000
t.c:3, t.c:12  --> 10
```

Figure 6.2: AFDO profile for a simple source code

For this example, if the foo function is inlined and bar is not, the function profile for the function a should resemble the one presented in Figure 6.2.

The instructions of the inlined function are not present on their own in the profile. They are present in the inline stacks of the caller function. If the foo function contained an inlined function, the inline stacks would get one level deeper. If a function is now always inlined but also called normally, it will be present both as a top-level function and inside other function inline stacks.

This example is ideal since the maximum number of counts have been recorded. In practice, the numbers would not be as accurate.

6.3 Implementation

The implementation consists of a C++ application processing the Gooda spreadsheets and producing an AFDO file.

The implementation needs GCC 4.7 and CMake 2.8 to build. C++11 is used in this application, its support must be activated. Several Boost libraries are also necessary.

One constraint has been to keep the converter as generic as possible. It means that, in the future, the converter should be able to process Gooda files and analyze them to do something else than converting them to AFDO (e.g. computing the difference of two sets of spreadsheets). The analyzer should also be able to generate AFDO format from another source than Gooda spreadsheets without changing too much code.

For that, the implementation is separated into three tasks:

1. Read and process the Gooda spreadsheets. It generates a data structure containing all the information of the spreadsheets.

2. Process the Gooda data structure and transform it into an AFDO-compatible data structure.
3. Generate the AFDO file from the AFDO data structure.

The application supports Gooda in two modes:

- Cycle Accounting mode
- Last Branch Record (LBR) mode

For the converter, the mode does not change the way the Gooda spreadsheets are read but changes the way they are converted to AFDO. The generated profiles will be slightly different depending on the mode.

Each task is detailed in the following sections.

6.3.1 Read Gooda files

The reading of the Gooda Spreadsheets is made in a very generic way into data structures. Each file consists of a list of lines. Each line is an array of cells. For performance reason, a cell is a simple pair of iterators pointing to the original line, not a string directly. The rows are converted to an integer or to a string only when they are accessed.

The CSV parser has been implemented especially for this project because no CSV parser fast enough has been found.

6.3.2 Cycle Accounting Mode

In this mode, the count of an element is the number of Unhalted Core Cycles reported by Gooda. It is the number of cycles spent during when the processor is not halted. This is a common measure in Cycle Accounting. Most of the time, the processor is processing and executing instructions, it is in Unhalted state. However, sometimes, the processor has to wait for another event, for instance an I/O operation and during this time, it is not doing anything, it is in halted state.

This mode is a bit simpler than the LBR one because the information is already present as expected by AFDO. Indeed, AFDO expects an instruction profile and Gooda reports the number of cycles for each assembly instruction.

For each line present in the assembly view, an inline stack is created. The count is the number of Unhalted Core Cycles of the assembly instruction. The number is multiplied by the multiplexing factor reported by Gooda.

If the same line is encountered several times in the assembly view, the count of this line is set to the maximum count found in the assembly view. Each time the line is encountered, the number of dynamic instructions is incremented by one.

6.3.3 LBR Mode

The difference between the two modes lies in the counter used as a profile count. In LBR mode, Gooda uses a special script using the LBR feature of the processor for precise sampling. With that, Gooda already computes the number of executions of each basic block, doing almost the same aggregation job as AFDO. In this mode, the instructions have a value of zero for their counter, only the basic blocks in the assembly view have a counter value. The number of basic block executions is reconstructed from the Branch Instruction Retired event and the LBR stacks.

It is not possible to give basic block profiles to AFDO, so the counters of all the instructions of a basic block are set to its number of executions.

For that, it is necessary to get a list of all the basic blocks of the function. This information is already provided by Gooda. All the basic blocks are extracted from the DWARF information and the counters are directly linked to the basic blocks in the assembly view. For each basic block, the list of the assembly instructions belonging to it are directly present after it in the spreadsheets.

Two different counters are used:

- The number of software instructions retired is used for the total execution count of the functions.
- The number of basic block executions is used as a counter for each instruction.

For each assembly instruction present in a basic block, an inline stack is created and its count is the number of execution of the basic block. Again, each time the same inline stack is encountered in the assembly view, its number of dynamic instructions is incremented.

6.3.4 Inlining

AFDO expects the profile to be filled by inline stack. If `bar()` is inlined into `foo()`, there will be at least one inline stack made of instructions from both `bar()` and `foo()` functions.

To generate this kind of profile, it is necessary to know for each dynamic instructions in a compiled function if this instruction comes from the function itself or from a call that has been inlined. Gooda provides this information in the assembly view. However, the information is limited to one level of inlining. In practice, an inlining chain can be of arbitrary length.

Gooda provides for each dynamic instruction the principal file and line of where does the instruction come from and in the case of an inlined function, the initial file and line inside the callee. AFDO expects one more information for the inlined function: its name. In some case, it is possible that the inlined function is also a hotspot function if called normally some other time. If it is not the case, the information is not available in Gooda.

There are several possible solutions to find the name of the inlined function:

1. Use `objdump` to extract the information using the address of the dynamic instruction.
2. Use `addr2line` to get the complete call chain for each instruction.
3. Extract the information by reading the DWARF debug symbols.
4. As the source line and file are known, parse the source file to find the inlined function name.

Item 4 has not been considered because it would be a very complex task to parse C/C++ syntax. Moreover, it would mean that the converter needs the source files, which is not very practical (even if Gooda already needs the source files). Item 3 would have been complicated as well to implement. Moreover, there are already tools performing this task.

Finally, it has first been decided to use `objdump` (Item 1). However, for performance reason, the initial implementation with `objdump` has been replaced with `addr2line` (See Section 6.5.1 for details).

As `addr2line` has an option to get the complete inline stack, it has been decided to use it to overcome the limitations of Gooda. The collection of the inline stacks is made in several steps:

```
1 int a = 0;
2
3 for(int i = 0; i < 1000; ++i){
4     a += i;
5 }
```

Figure 6.3: Discriminators example

1. All lines of all assembly spreadsheets are searched for inlined functions. For that point, the Gooda feature is used to identify lines coming from an inlined function. In these cases, Gooda indicates the file where does the inlined function come from. When this file is not empty, the address of the instruction is stored for the next step. It would have been possible to run the requests on any assembly instructions, but using the information from Gooda makes it faster as there are less instructions to be processed by `addr2line`.

2. Once all the interesting addresses are collected, `addr2line` is used on each ELF file containing inlined functions.

3. The result of `addr2line` is parsed to find all the necessary information to construct the inline stack (the function name, the file name, the source line number and the discriminator (See Section 6.3.5)).

The inline stacks are then used during the annotation of the functions. Each time the same inline stack is encountered in the assembly view, the number of dynamic instructions of the inline stack is incremented and its counter is set to the maximum between the previous and the new values.

6.3.5 Discriminators

A discriminator is a debug information from the DWARF standard enabling to distinguish several statements that are on the same line of source code. Figure 6.3 presents the most common example where discriminators are important. The line 3 will result in three different sets of instructions each denoted by different discriminator. The first one (discriminator=0) indicates the initialization of `i`, the second (discriminator=1) denotes the test and jump of the loop and the final one (discriminator=2) indicates the increment of `i`.

In this case, it is very important to associate the correct DWARF discriminator to the correct set of assembly instructions.

To correctly associate the profile sample to the GIMPLE instructions, the AFDO format expects the DWARF discriminator for each source position. However, Gooda does not provide this information. It has been necessary to find another tool to complete the information provided by Gooda.

Since its version 2.23.1, the binutils sets of tools support DWARF discriminators. `addr2line` has been chosen to get this information. The collection of the discriminators is made in several steps:

1. The addresses of all the assembly instructions present in the assembly view of the Gooda spreadsheets are collected for each executable or shared library.
2. One call to `addr2line` is made for each ELF file with the list of addresses.
3. The result of the command is parsed to extract the discriminator for each address. The discriminators are stored in a cache.
4. During the normal generation of the AFDO profile, each time an instruction is encountered, its discriminator is taken from the cache using its address as a key.

With that solution, the overhead is very small because only one call to `addr2line` is necessary for each ELF file.

One important point to take into account is that DWARF does not support discriminators to be set for the inline stack of an instruction. However, there is a bug in `addr2line` 2.23.1, causing the discriminators of the whole inline stack to be set to the value of the discriminator of the last callee line. It is important to take into account only the source discriminator and not the ones of the inline stack. The bug has been sent to the developer responsible of `addr2line` development and will be fixed.

As this solution works only with binutils 2.23.1, the collection of the discriminators is only made if the user specified the `--discriminators` option.

The request for discriminators can work on a very large numbers of addresses. To avoid creating too long commands, the addresses are put in a file that is then used by `addr2line`. Using very long requests from C++ caused some problems collecting the results, these problems disappear when writing the addresses to a file before calling `addr2line`.

6.3.6 Function names

When a language supports features like function overloading or namespaces, the name of a function is not enough to uniquely identify it. For this reason, in languages like C++, name mangling is used to create unique names for functions based on their names, the enclosing scope, the types of the parameters, etc. It means that each function has two names.

Moreover, when it comes to GCC and the debug symbols, the problem is even more complicated as each function has three names:

- The unmangled name: It is the human-readable version of the function name.
- The mangled name, or assembler name: It is the name used in the assembly code to uniquely identify the function.
- The BFD name, or short name: This name is used by GCC in the debug symbols to identify inlined functions.

AFDO expects the name of each top level function to be its assembler name. For the function names inside the inline stacks, the name can be either the assembler name of the BFD name. AFDO generates a hash table matching the BFD names to the assembler names based on the information given by GCC on the current functions.

Unfortunately, Gooda reports only unmangled names, which are useless in this case.

One solution would have been to mangle the names from the unmangled names, however, this is a complex task and rather heavy. Moreover, in some cases, the information reported by Gooda is not complete enough to recreate the mangled names. Indeed, when the names are too long, Gooda cuts them to a certain number of characters.

In the beginning, it has been decided to use `addr2line` to get the mangled function names. This works well most of the time, but sometimes `addr2line` reports a BFD name instead of the assembler name. This is a known bug of `addr2line`.

For that, it has been decided to use `objdump` to get the mangled function names. This is made in several steps:

1. For each hotspot function, get the address of the beginning of the function and the ELF file the function is compiled into.
2. For each ELF file, use `objdump` to get the complete list of symbols.

3. Parse the result of `objdump` to get the assembler name of each symbol.

4. For each hotspot function, get the assembler name from the cache filled in the previous step and replace the current name with it.

With these steps, all the functions have a correct assembler name with at most one call to `objdump` for each ELF file of the application. As only the symbol table is read, this is very efficient. Another advantage of using directly the symbol table instead of the disassembly (in which the information could have been collected too) is that inlining does not need to be considered.

6.3.7 Export To AFDO

Considering what have been studied in the last sections, the AFDO profile can be generated with the algorithm presented in Figure 6.4.

The routine responsible of collecting the basic blocks gets the counter of the basic blocks, if any and the assembly instructions that are parts of a given basic block (denoted by a range).

The annotate function of each mode works in the same manner. The difference is how a count is given to each inline stack.

Once all the functions have been annotated by either one of the two modes and the function profile is complete, the other operations are performed independently of the current mode.

The working set is generated in a second time when information about all the functions has been collected. The first step is to generate a histogram representing for each count the number of dynamic instructions that have been counted this number of times. Then, the histogram is traversed in descending order. During the traversal, the accumulated count is computed and each accumulated count is assigned to the correct position into the working set.

Gooda performs system wide analysis and so can collect functions in system processes. The process view contains the list of processes running during profiling. No data is extracted from the process view spreadsheets. Only the process running the profiled application is interesting for the profile.

The length of the different sections is computed once every data has been imported. It is calculated in a unit of four bytes.

```
for all hotspot function $f$ do
    if $f$ has a counter value then
        $function \leftarrow$ new AFDO function
        $function.elf\_file \leftarrow$ path to the containing ELF file
        $function.total\_count \leftarrow$ total from the hotspot function view
        if $valid(function)$ then
            Store the function
        end if
    end if
end for
Set the assembler names for each function
Fill the inlining cache
Fill the discriminator cache
for all stored function $function$ do
    $basic\_blocks \leftarrow collect\_bb(function)$
    $function.entry\_count \leftarrow$ count of the first basic block
    $function.file \leftarrow$ file of the first instruction of the first basic block
    if LBR then
        $lbr\_annotate(function, basic\_blocks)$
    else {Cycle Accounting}
        $ca\_annotate(function, basic\_blocks)$
    end if
end for
Remove functions with no counts
Use relative file names
Fill the string table
Compute the working set
Compute the sizes of the sections
```

Figure 6.4: AFDO general algorithm

Once the necessary data has been imported into the AFDO data structure, the AFDO file is generated. The generation is made using the GCOV format. However, the GCOV library has not been used, for reasons explained in Section 9.5. The GCOV features have been rewritten directly in C++, keeping the same format

6.3.8 Other features

The application is based on the Gooda analyzer to collect the data. This tool is not the most user-friendly application that is. It is necessary to know the processor model to choose the correct collection script, then run it and finally run Gooda. Both applications have to be run as superuser. To ease this process, the converter can automatically execute these steps. If the user uses the `--profile` option, the converter will run the given application with the Gooda script, analyze the data with Gooda and finally convert the spreadsheets to an AFDO file. The processor model is automatically discovered and the used script is chosen based on this information. If the processor model cannot be found, the user can specify it with the `--model` option. If the `--lbr` option is used, all the operations are made in LBR mode.

The application can also output all the extracted information in a human-readable format if the user specifies one of the `--dump` or `--full-dump` options.

As Gooda performs a system-wide analysis, the list of hotspot functions can contain functions from other processes or from the Linux Kernel for instance. Generally, there are few samples from other processes. However, in some environment, it is possible that the other processes are present a lot in the Gooda spreadsheets. Generating profile for these functions is useless because that is not the functions that will be optimized and it can be costly. For these reasons, the converter can filter the functions by process. If the user specifies the `--filter` option, only functions of the hottest process are taken into account. If the user specifies a specific process with the `--process` option, only hotspot functions of the given process are exported to the AFDO file.

In order to be able to compare AFDO files, a feature has been added to the converter to read AFDO files. The converter can read any AFDO binary file that has been produced by either this application or by the Google application. The reading is made the same way it is done in the GCC AFDO patch. The GCOV functions to read files have again been rewritten in plain C++. When the AFDO file is read, it is dumped to the console.

The converter is also able to perform basic comparisons between two AFDO files.

It reads both files and then extracts the differences between the two profiles. The order of the hotspot functions is compared. The differences between their counter values is also computed. It verifies that each inline stack is present on both profiles and indicates which inline stack are lacking in one of the profiles. The search is able to handle BFD name or assembler names in the inline stacks. For each inline stack present in both profiles, the difference between the counter values is also computed.

6.4 Tests

The converter has mainly been tested with the SPEC test suite by verifying its accuracy against traditional PGO, see Section 6.6.

However, the converter has also been tested with several unit tests. The advantage of the unit tests compared with the benchmark itself is that it runs much faster and automatically test several points of the profiles while the benchmark only gives the accuracy of the tool chain.

The unit tests have been developed with the Boost Test Library[3]. This framework has been chosen because Boost was already used for the converter itself. Moreover, this is a very complete framework that fulfills all the needs of the unit test suite.

Several test cases have been implemented:

- A C90 program computing a count from a dynamic array. One function is inlined directly into main.
- A C++ version of the previous program. The differences are in the function names (C++ use mangling) and in the function inlined for the printing.
- A C++ program with inheritance and several functions making computations on a dynamic array.
- A C++ program with a high number of inlining (up to nine levels) to verify that all levels are correctly discovered. The functions are located in different source files to verify that the files of the inlined functions are found. This test case also includes several template instantiations to verify the names of the functions.
- A Fortran 90 program computing the sum of the areas of a billion of triangles.

[3] www.boost.org/libs/test/

For each test case, the source file has been compiled on the LBNL server with GCC. Then, the executable has been profiled using one of the Gooda script. Finally, the spreadsheets have been generated with Gooda. The spreadsheets have been collected in both UCC and LBR mode. The compiled programs and the two sets of spreadsheets are saved in the repository.

Each unit test defines several points to be tested:

- The number of hotspot functions
- The number of hotspot processes
- The total count and entry count of the functions
- The function names (must be assembler names) and file names
- Several inline stacks are tested specifically
  - The count must match the count in the spreadsheets
  - The name must be in the correct format (assembler name or BFD name)
  - The discriminators must be set correctly
  - The inlining information must be correct

As the samples are simple, the test cases have been written by reading the spreadsheets by hand to be sure that the data were correct.

The tests helped to found some problems with the handling of inlining with `addr2line` and helped avoid regressions when changes were made to the converter.

6.5 Performances

As the spreadsheets generated by Gooda can become quite large, it has been important to keep in mind the performance of the conversion during the implementation.

To see the time necessary to convert spreadsheets, the converter has been tested on the spreadsheets generated for different programs. To stress both Gooda and the converter, the sample period of LBR mode has been set to 40,000 instead of 400,000.

The performances of two modes are tested:

- ucc: Cycle Accounting mode.

- lbr: Last Branch Record mode.

Several cases are tested:

- *gcc-google-eddic*: The compilation of eddic by GCC. The sizes of the perf.data files are 167MB for CA and 4.4GB for LBR.
- *gcc-google-converter*: The compilation of the converter by GCC. The sizes are 4.0MB and 109MB.
- *eddic-assembly*: eddic compiling assembly.eddi. The sizes are 203KB and 4.3MB.
- *eddic-list*: eddic compiling linked_list.eddi. The sizes are 194KB and 3.0MB.
- *converter-gcc-ucc*: The converter working on the data produced in Cycle Accounting mode at Item 6.5. The sizes are 396KB and 13MB.
- *converter-gcc-lbr*: The converter working on the data produced in LBR mode at Item 6.5. The sizes 436KB and 13MB.

All the results have been collected on a server at LBNL. This computer is running a Gentoo operating system on top of Linux 3.6.11 kernel. The processor is an Intel® Xeon™ E5-2650, running at 2 GHz. The server was dedicated to the project, no other user was using it. The converter has been compiled with the following options: -g -O2 -march=native with GCC 4.7. The output of the tools is redirected to `/dev/null`.

Each test is run five times and the minimum time of those five executions is taken into account.

Several performance improvements have been necessary before the results have been collected because the application was not fast enough. The implemented changes are presented in Section 6.5.1. Figure 6.5 presents the obtained results.

The first thing that can be seen from the results is that the converter performs more or less the same in ucc or lbr mode, except for the eddic benchmarks. In that case, ucc is slower than lbr by a factor two. In this case, Gooda reported more functions in the ucc version than the lbr version, giving more work to the converter.

The time necessary to convert the spreadsheets to the AFDO file is not negligible. Indeed, on the eddic cases it takes up to six seconds to complete. The interesting point is that the time necessary to generate the AFDO file is not directly linked to the profiling time. Indeed, it takes almost the same time to convert the profiles of

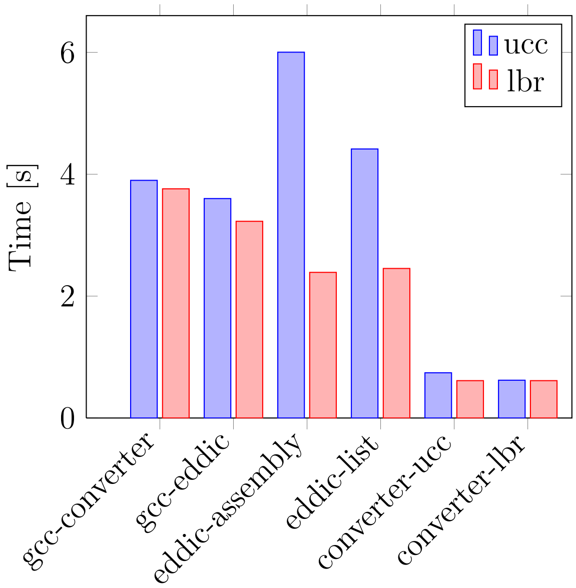


Figure 6.5: Converter Performances

the two GCC compilations, but the compilation of eddic took 45 times longer than the compilation of the converter. In both cases, Gooda reported almost the same hotspot functions, which is why the times are so close. The fact that Gooda reports only a small subset of the total number of functions helps keep the conversion time reasonable.

For the profiles of the converter itself, the time is really low, it takes less than half a second to convert them.

The converter has been profiled to see what was taking the most important part of the time.

1. In average, more than 60 percent of the time is spent in calls to `addr2line` and `objdump` to fill the inlining and discriminator caches and to get the mangled names of each function. `addr2line` takes about 80 percent of this time.

2. About 20 percent of the time is spent in processing the spreadsheets.

3. Only 15 percent of the time is spent converting the Gooda format to the AFDO profile format.

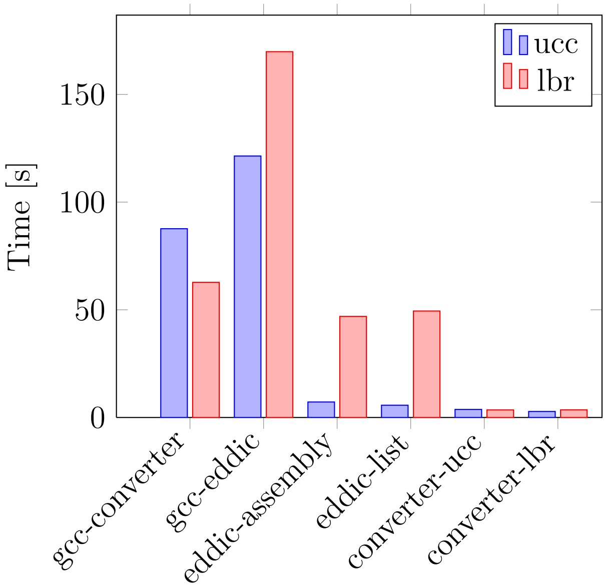


Figure 6.6: Gooda Performances

4. The remaining time is spent generating the file itself.

The time necessary for Gooda to generate spreadsheets has also been tested. The tests are made on the same cases and with the same computer configuration. Figure 6.6 presents the time necessary for Gooda to generate spreadsheets.

The time necessary for Gooda is always higher than the time necessary for the converter. It is clearly not negligible. Indeed, when generating the spreadsheets of the compilation of eddic by GCC, it takes about two minutes and a half, which is relatively high. In that case, the overhead is OK because the compilation time was about 40 minutes. On the other hand, generating the spreadsheets for the profile of the compilation of the converter took one minute and a half in UCC mode and the compilation itself took less than one minute. It means that in some case, generating the spreadsheets takes longer than the execution itself.

The other profiles were generated quite fast. Except for the LBR profiles of eddic. In that case, Gooda seems to have problems generating this profile. It issues a lot of warnings. One possible reason that has been found is that the function names are very long due to the heavy usage of templates in eddic and that Gooda has issues handling them. This is a point that should be improved in the future in Gooda itself.

In general, the time necessary for the conversion is highly depending on the Gooda output and especially the numbers of hotspot functions and total size of the spreadsheets to read. It takes a long time to parse them. The more spreadsheets there are, the longer the queries to addr2line will take. For Gooda, the size of the profile is important, but it is not always directly linked. Indeed, in some cases, it seems that for some reason Gooda has problems with C++ and LBR together. It could be that as the LBR profile is more precise, Gooda has more functions to digest or that the LBR samples handling is not always optimal.

To conclude, the time necessary for the converter to generate the AFDO file is generally much lower than the time necessary to generate the spreadsheets by Gooda.

When the two tools are put together, the time is not negligible, it can take minutes to generate the AFDO file from the samples generated by perf. One may argue that in practice, it should take less because the sampling period has been reduced. It is only true for the LBR mode. The Cycle Accounting mode sampling periods have not been changed. On the other hand, for large applications, the runs should be longer to cover the most important hotspots and so the time to generate the AFDO profile should not be as important compared to the profiling time.

6.5.1 Optimization

From the beginning of the benchmarking, it has been clear that the calls to objdump (to discover the names of the inlined functions) were very costly. For a binary with many inlined functions in the hotspot functions, it accounted for more than 95 percent of the total time.

The first thing that has been done to improve this state has been to tune the options passed to objdump. The time spent in objdump has been decreased by a factor of two, by selecting only the .text section of the ELF file. Even if this is a big increase in performance, it was still slower than our expectations.

Once it was clear that objdump could not be made faster, the possibilities considered at the first place (See Section 6.3.4) were considered again. addr2line has been considered first because it is the only one that would not require major efforts to use instead of objdump.

At first sight, addr2line is not faster than objdump. However, addr2line can generate the necessary data for several addresses at the same time. It turns out that addr2line takes approximately the same time to collect data about one address or hundred. The long time is not made to collect the data for the addresses but to collect the symbols at the beginning and it is only made once.

With that property, addr2line is a very interesting candidate. The code was rewritten by adding a new pass before the functions are annotated by either LBR or CA mode. This pass collects all the instructions coming from inlined functions. Then, a single call to addr2line with all the addresses is done, the results are parsed and the inline stacks are cached. Then, the normal execution is made just like before, but now the inline stacks are taken from the cache directly. It resulted in a very large increase in performances, up to 45 times faster in the case of the gcc-google-converter benchmark case.

After this optimization, the converter has been considered fast enough for this project.

6.6 Results

This section presents the results obtained with the implementation presented in this chapter.

All the results have been collected on a server at LBNL. This computer is running a Gentoo operating system on top of a Linux 3.6.11 kernel. The processor is an Intel® Xeon™ E5-2650, 2 GHz. It is an Intel® Sandy Bridge microarchitecture. The server was dedicated to this project, no other user was using it. The CPU Frequency Scaling feature of the Linux Kernel has been disabled because it caused large variations on the results. By disabling it, the standard variation of some benchmarks has been reduced by two orders of magnitude.

The compiler used is the Google GCC 4.7 branch[4] with the AFDO patch.

Different versions are compared:

- base: Optimized executable. The benchmark is compiled with the following options: -g -O2 -march=native

- instr: GCOV instrumentation. The benchmark is compiled in two pass. The first pass is compiled with the same options as base, plus -fprofile-generate and -fno-vpt. The executable is then trained and finally compiled again with the same options as base plus -fprofile-use and -fno-vpt.

[4] http://gcc.gnu.org/viewcvs/branches/google/gcc-4_7/

- `ucc`: Our implementation in Cycle Accounting mode. The benchmark is first compiled once with the same options as `base`. Then, it is trained while monitored by `perf`. After that, Gooda is run to generate spreadsheets. Then, the converter generates an AFDO file from the spreadsheets. Finally, the program is compiled using the same options as `base` plus -fauto-profile. The event that is sampled is UNHALTED_CORE_CYCLES with a period of 2'000'000 events.
- `lbr`: Same options than `ucc` except that the toolchain runs in LBR mode. The event that is used is BRANCH_INST_RETIRED with a sampling period of 400'000 events.

The converter has been run using the –discriminators and –quiet options and using `addr2line` 2.23.1.

Gooda has been especially optimized for small programs. Gooda uses some cutoffs to limit the number of functions that are reported. These cutoffs have been modified to include as many functions as possible. For applications big enough, this should not be a problem as there are much more functions, but in small programs, each function is important. This also increases the time to generate the spreadsheets as more functions are included. This configuration is not usable on large programs, it took more than half an hour to generate the spreadsheets for a GCC profile with this configuration. Moreover, the scripts have been modified to run `perf` in quiet mode because the benchmark suite does not support any output during the execution.

The results were collected on the SPEC CPU 2006 V1.2 benchmark suite[5]. This benchmark suite has been chosen for several reasons. It contains a complete suite of very different benchmarks. The benchmarks are of several types, CPU-intensive, memory-intensive, floating points or integers. Moreover, it is the benchmark suite of choice for compiler optimizations benchmarking.

It has been necessary to select a subset of the benchmarks. Indeed, each benchmark is run three times plus the training phase. Each benchmark has to run with four different configurations. With the chosen subset, it already takes almost a day to complete.

[5] `http://www.spec.org/cpu2006/`

The following subset has been considered to collect the results:

- `401.bzip2`: A compression program written in C. The benchmark has been separated in two different data sets: input.program and input.combined.
- `410.bwaves`: A numerical simulation of blast waves written in Fortran.
- `433.milc`: A simulation of lattice gauge theory written in C.
- `450.soplex`: A linear program solver written in C++. Two different data sets have been used: ref and pds-50.
- `453.povray`: A ray-tracer written C++.
- `454.calculix`: A finite element code for linear and nonlinear three-dimensional structural applications written in C and Fortran.
- `458.sjeng`: An artificial intelligence playing chess, written in C.
- `462.libquantum`: A quantum computing library written in C.
- `464.h264ref`: A video compression algorithm written in C. Three different data sets have been used: forebase, foremain and sss.
- `465.tonto`: A Fortran calculator in the field of quantum crystallography.
- `470.lbm`: A simulation of incompressible fluids in 3D, written in C.
- `471.omnetpp`: A C++ simulation of discrete event over an Ethernet network.
- `473.astar`: Implementation of the A* path-finding algorithm, written in C++. The benchmark has been separated in two benchmarks with different inputs: big lakes and rivers.
- `483.xalanbmk`: An C++ XSLT processor transforming XML into HTML.

These benchmarks have been chosen to have several programming languages and to have both integer and floating point benchmark. They also have been chosen to represent several important fields of science (Physics, Artificial Intelligence, Compression, . . .).

benchmark	base	instr	ucc	lbr	I gain	U gain	L gain	U / I	L / I
astar.bigLakes	37.56	41.93	40.62	42.63	11.64	8.15	13.51	70.05	116.11
astar.rivers	21.18	23.42	23.77	24.08	10.58	12.26	13.69	115.89	129.41
bwaves	19.00	19.69	19.01	18.74	3.66	0.05	-1.35	1.44	0.00
bzip2.combined	98.60	100.33	100.18	100.01	1.75	1.60	1.44	91.37	81.93
bzip2.program	71.33	73.77	72.97	72.33	3.42	2.30	1.41	67.16	41.11
calculix	8.46	8.47	9.13	8.57	0.14	7.87	1.29	5550.00	908.33
h264.forebase	267.74	296.78	275.60	287.05	10.85	2.93	7.21	27.06	66.50
h264.foremain	348.76	369.22	355.19	364.95	5.87	1.85	4.64	31.45	79.17
h264.sss	39.92	41.58	40.19	41.31	4.14	0.66	3.48	15.90	84.04
lbm	30.05	33.31	32.47	31.54	10.88	8.07	4.99	74.15	45.82
libquantum	45.58	54.13	45.22	49.41	18.77	-0.79	8.40	0.00	44.78
milc	17.79	18.57	18.48	18.47	4.42	3.90	3.85	88.30	87.02
omnetpp	17.58	18.18	18.09	18.05	3.36	2.85	2.63	84.94	78.34
povray	24.67	29.14	27.49	27.70	18.10	11.42	12.28	63.12	67.82
sjeng	18.00	18.64	17.71	18.38	3.54	-1.66	2.10	0.00	59.40
soplex.pds-50	57.27	58.36	56.43	58.02	1.90	-1.46	1.30	0.00	68.32
soplex.ref	67.19	71.07	69.07	69.95	5.77	2.80	4.10	48.54	71.11
tonto	17.83	18.28	17.94	18.13	2.50	0.61	1.69	24.44	67.49
xalancbmk	24.56	24.22	24.67	25.30	-1.39	0.44	3.03	∞	∞
mean					6.31	3.36	4.72	53.26	74.80

Table 6.1: Sampling-based PGO results (Numbers are SPEC CPU 2006 scores, higher is better). The benchmarks have been trained with the training data set. I: Instrumentation, U: Cycle Accounting, L: LBR. The gains are displayed in percent. Arithmetic mean used for average gain.

Table 6.1 shows the results obtained.

The results are very interesting. Indeed, LBR mode achieves about 75 percent of the gains that are obtained with instrumentation-based PGO. Cycle Accounting is less effective, achieving only 53 percent of the gains. This is an expected result as the LBR event is much more accurate than Unhalted Core Cycles.

In some benchmarks, Sampling-based PGO is even more effective than the traditional approach. The two `astar` benchmarks are very interesting with LBR achieving from 116 to 129 percent of the traditional PGO gains. This is even more the case for the `calculix` and `xalancbmk` benchmarks where instrumentation gains are very poor and sampling versions perform very well.

On the other hand, there are also benchmarks for which the toolchain do not perform very well. For instance, on `bwaves`, Cycle Accounting achieves only one percent of the possible gains and LBR makes the resulting executable even worse.

On several benchmarks, Cycle Accounting is making the performance worse, while LBR is much better. Nonetheless, there are also cases where it is the contrary. For instance, on `calculix`, Cycle Accounting is several times better than the LBR version. Several other benchmarks are also presenting the same behavior, but with less difference between the two versions.

These results have been obtained on the training data sets. All the SPEC CPU 2006 benchmarks contain different data sets. The score is not obtained on the same data set that is used for training. The training data set is generally smaller than the reference data set. The correspondence between the training data set and the reference data set has been calculated to be from 86 percent to 100 percent [Gove2007].

To see the difference, the benchmarks were also tested by training with the reference data set. Table 6.2 shows the detailed results obtained with this approach.

The first interesting point is that the PGO results are all better with the training on the same data set than the final execution itself. Instrumentation and Cycle Accounting modes are performing about 22 percent better and LBR is performing 37 percent better. LBR seems to be more sensitive to the input data set. On average, LBR mode is now achieving 84 percent of the instrumentation-based PGO gains.

On three benchmarks (`libquantum`, `sjeng` and `bwaves`), Cycle Accounting mode is still performing very bad. On the other hand, there are still some benchmarks for which it is performing better than LBR, but the margin is now smaller. The reasons for this behavior have not been found so far. It is possible that it comes from a problem in the toolchain with the handling of LBR profiles, either in Gooda or in the converter itself.

For two benchmarks, `calculix` and `xalancbmk`, the results are very different than when training with the regular training data set. It seems that in those two cases, the training data sets are not very representative of the reference data set.

The results on the `bwaves` benchmark are not good neither for Cycle Accounting nor LBR mode. This has been confirmed by Google with their internal toolchain. It comes from the fact that AFDO is especially optimized for C++ applications.

Table 6.3 presents the results with the C++ benchmarks only.

On C++ benchmarks, both sampling versions are achieving high gains. Cycle Ac-

benchmark	base	instr	ucc	lbr	I gain	U gain	L gain	U / I	L / I
astar.bigLakes	37.57	41.12	42.20	42.41	9.44	12.32	12.89	130.50	136.51
astar.rivers	21.06	23.68	22.85	24.07	12.42	8.49	14.29	68.35	115.02
bwaves	19.09	20.20	19.11	19.42	5.85	0.15	1.75	2.51	29.93
bzip2.combined	98.57	100.60	99.73	99.60	2.06	1.17	1.05	56.86	50.86
bzip2.program	71.25	73.26	73.31	72.27	2.82	2.89	1.43	102.39	50.75
calculix	8.46	9.23	9.16	9.17	9.07	8.28	8.43	91.26	92.96
h264.forebase	268.01	296.68	279.63	288.46	10.70	4.34	7.63	40.53	71.31
h264.foremain	350.49	378.67	361.36	370.31	8.04	3.10	5.66	38.57	70.35
h264.sss	39.83	42.66	41.25	42.18	7.12	3.56	5.89	50.07	82.75
lbm	30.12	31.60	32.34	31.74	4.92	7.38	5.39	149.83	109.44
libquantum	44.95	52.59	44.53	50.49	16.99	-0.95	12.32	0.00	72.50
milc	17.97	18.72	18.76	18.54	4.16	4.40	3.14	105.61	75.53
omnetpp	17.62	18.53	18.09	18.32	5.13	2.62	3.97	51.11	77.43
povray	24.81	29.16	26.33	28.10	17.51	6.13	13.24	34.98	75.58
sjeng	18.01	18.26	17.67	18.41	1.39	-1.85	2.26	0.00	162.15
soplex.pds-50	56.97	58.45	58.08	58.22	2.59	1.96	2.20	75.53	84.95
soplex.ref	66.88	71.78	69.16	70.56	7.32	3.40	5.49	46.49	75.02
tonto	17.79	18.27	17.95	18.14	2.69	0.89	1.96	33.19	72.86
xalancbmk	24.52	28.65	27.15	28.04	16.85	10.73	14.38	63.66	85.35
mean					7.74	4.16	6.49	53.71	83.87

Table 6.2: Sampling-based PGO results for all benchmarks (Numbers are SPEC CPU 2006 scores, higher is better). The benchmarks have been trained with the reference data set. I: Instrumentation, U: Cycle Accounting, L: LBR. The gains are displayed in percentage. Arithmetic mean used for average gain.

benchmark	base	instr	ucc	lbr	I gain	U gain	L gain	U / I	L / I
astar.bigLakes	37.57	41.12	42.20	42.41	9.44	12.32	12.89	130.50	136.51
astar.rivers	21.06	23.68	22.85	24.07	12.42	8.49	14.29	68.35	115.02
calculix	8.46	9.23	9.16	9.17	9.07	8.28	8.43	91.26	92.96
omnetpp	17.62	18.53	18.09	18.32	5.13	2.62	3.97	51.11	77.43
povray	24.81	29.16	26.33	28.10	17.51	6.13	13.24	34.98	75.58
soplex.pds-50	56.97	58.45	58.08	58.22	2.59	1.96	2.20	75.53	84.95
soplex.ref	66.88	71.78	69.16	70.56	7.32	3.40	5.49	46.49	75.02
xalancbmk	24.52	28.65	27.15	28.04	16.85	10.73	14.38	63.66	85.35
mean					10.04	6.74	9.36	67.12	93.22

Table 6.3: Sampling-based PGO Results for C++ benchmarks (Numbers are SPEC CPU 2006 scores, higher is better). The benchmarks have been trained with the reference data set. I: Instrumentation, U: Cycle Accounting, L: LBR. The gains are displayed in percentage. Arithmetic mean used for average gain.

counting achieves 67 percent of the instrumentation gains and LBR achieves 93 percent of the gains.

These results are very promising. With a small overhead, LBR is able to achieve 93 percent of the gains obtained with the instrumentation approach of PGO.

This section presented some results that can still be improved, especially when Cycle Accounting performs better than LBR. In theory, this should not happen. It is possible that it comes from problems in a part of the toolchain. Once fixed, it is possible that the results could be even better. Once AFDO has improved support for other languages, it will be possible to achieve very good results on average for all the SPEC benchmarks. Moreover, on some of the benchmarks, the profiles generated by Google with their internal toolchain have been more effective than the profiles generated with this toolchain, so there is still room for improvement.

The results presented in this chapter are not “SPEC-reportable”. Indeed, to allow training with the reference data set, the SPEC tree has been switched to a development tree. Moreover, some benchmarks have been separated between different data sets and this is against the rules of SPEC for reportable results.

6.6.1 Overhead

Sampling is especially interesting because it simplifies the toolchain that is used to perform PGO. Profiling the executable is generally much faster than running the instrumented version. To estimate this overhead, the same tests have been run again, but this time the tested executable is the instrumented or profiled one.

Again, four versions are tested:

- `base`: Optimized executable. The benchmark is compiled with the following options: -g -O2 -march=native
- `instr`: GCOV instrumentation. The program is compiled with the same options as `base`, plus -fprofile-generate and -fno-vpt.
- `ucc`: Cycle Accounting profiling. The program is compiled with the same options as `base` and then run under the Gooda collection script.
- `lbr`: Same options as `lbr` except that the collection script is using LBR event.

Table 6.4 shows the overhead of the different versions.

The overhead of instrumentation is high, but no as much as expected, 16 percent on average. The highest overhead is 53 percent for the `povray` benchmark.

As expected, the overhead of sampling is lower than the overhead of the instrumentation. The overhead of LBR is very low, 1.06 percent on average. In several benchmarks, the overhead is less than 1 percent. In average, the overhead is almost 16 times lower than instrumentation.

Even if less than the overhead of instrumentation, the overhead of the Cycle Accounting version is important, only 1.6 times lower than instrumentation, almost 10 percent on average. This version collects much more events than the LBR version, more than 60 on Intel® Sandy Bridge. For several benchmarks, the Cycle Accounting version is even slower than the instrumented version.

As only the number of Unhalted Core Cycles is used in the Cycle Accounting version to generate the profile, it would be interesting to add support to a new script in Gooda collecting only the Unhalted Core Cycles. As the event is sampled with a long period, this should reduce the overhead by a large margin.

An interesting point is that the overhead of instrumentation is highly variable. The overhead ranges from -3 percent to 53 percent. The overhead of sampling is much

benchmark	base	instr	ucc	lbr	I over	U over	L over
astar.bigLakes	187.19	209.29	206.30	188.04	11.81	10.21	0.46
astar.rivers	287.97	321.41	313.67	290.47	11.61	8.92	0.87
bwaves	712.14	697.22	779.52	720.48	-2.09	9.46	1.17
bzip2.combined	97.92	104.63	107.12	98.59	6.85	9.40	0.68
bzip2.program	135.51	146.37	147.99	136.61	8.01	9.21	0.81
calculix	973.41	987.17	1063.60	981.27	1.41	9.27	0.81
h264.forebase	82.51	109.41	90.57	83.46	32.60	9.76	1.14
h264.foremain	63.09	79.56	69.31	63.99	26.12	9.86	1.43
h264.sss	555.04	693.16	608.98	561.63	24.88	9.72	1.19
lbm	428.47	429.60	472.57	429.58	0.26	10.29	0.26
libquantum	463.88	449.05	509.31	473.55	-3.20	9.79	2.08
milc	503.85	535.57	558.84	509.84	6.30	10.92	1.19
omnetpp	337.00	452.85	379.28	340.07	34.38	12.55	0.91
povray	186.20	285.59	201.04	188.29	53.38	7.97	1.12
sjeng	592.29	748.70	646.59	600.93	26.41	9.17	1.46
soplex.pds-50	144.53	152.15	163.58	146.04	5.27	13.18	1.05
soplex.ref	124.17	132.12	136.00	125.08	6.40	9.53	0.73
tonto	473.77	497.74	514.79	480.02	5.06	8.66	1.32
xalancbmk	281.17	419.47	313.00	285.49	49.19	11.32	1.53
mean					16.03	9.96	1.06

Table 6.4: Profile-Guided Optimization Overhead Detailed Results (Numbers are seconds, lower is better). The overhead columns are percentages. Arithmetic mean used for average overhead.

more stable. The overhead of Cycle Accounting ranges from 8 percent to 13 percent and LBR ranges from 0.3 percent to 2 percent. The cost of instrumentation is highly depending on the type of code that is executed. For instance, a code that is branch dominated will have a much higher overhead than a code that is computation dominated because only edges are instrumented. On the other hand, the overhead of sampling depends on the number of events and the number of times the buffer has to be written to disk.

It could seem incorrect that some instrumented executables are actually faster than the optimized version. Even if rare, it may happen for some reason. After the instrumentation instructions are inserted in the program, the program is still optimized. These few instrumentation instructions can alter the optimization decisions. For instance, it is possible that a loop is not unrolled because of these instructions, resulting in smaller code. In some cases, the differences in the optimization decisions can cause the generated instrumented executable to be faster.

In this benchmark, only the overhead of `perf` has been measured, the time necessary to convert the Gooda spreadsheets to AFDO is not taken into account. The overhead of the converter and Gooda is discussed in Section 6.5.

6.7 Cache misses

Once the conversion to AFDO has been finished and tested, support for cache misses has been implemented. This support needs changes in several places:

1. GCC needs to store the cache misses for each basic block in its internal data structures.

2. The AFDO patch needs to extend its data structures and file format to handle cache misses in basic blocks.

3. The converter needs to extract the data from the Gooda spreadsheets and generate the new data in the AFDO profile.

The first two points have been implemented in a single patch for GCC.

6.7.1 GCC

At the present time, the profile information is stored directly in each `basic_block` as several member fields. It is not possible to transparently add new counters without modifying the structure.

As GCC does not support storing profile information for a statement, but only at a basic block level, the number of cache misses will be stored directly inside the `basic_block` structure. To store it, an `unsigned long` is used. This type represents an unsigned integer of 64 bits, allowing numbers up to $2^{64} - 1$, enough to represent any number of cache misses in an application. No new information is necessary on the edges.

6.7.2 AFDO

The first change that is necessary in AFDO is to add support for cache misses in the file format and in the data structures representing the file. The new counter is present in each stack. It is represented by a counter type (64 bits unsigned).

After that, it has been necessary to edit the functions getting information from a stack to include the number of cache misses.

For a basic block, the number of cache misses is the sum of the cache misses of all its instructions. The number of cache misses of a basic block is set during the annotation of the Control Flow Graph (CFG) by the AFDO pass.

6.7.3 Converter

In the converter itself, few changes are necessary. The new field must be added to the AFDO data structures and then used in the generation of the AFDO file.

To maintain compatibility with the original patch, the cache misses are only generated in the profile if the user specifies the `--cache-misses` option.

Gooda does not directly provide the number of cache misses. Instead, Gooda proposes a more advanced notion, called Load Latency. It gives an indication about how much time is spent during loads. Only the time that could have been avoided is computed. Load Latency is based only on loads because in practice Store Latency will rarely stall the pipeline. It is a metric based on all the levels of cache as well as the memory, making it more accurate than simply using the number of cache misses.

$$
\begin{aligned}
\text{load\_latency} = {} & 6 * \text{mem\_load\_retired:l2\_hit} \\
& + 52 * \text{mem\_load\_retired:l3\_unshared\_hit} \\
& + 85 * (\text{mem\_load\_retired:other\_core\_L2\_hit\_hitm} - \text{mem\_uncore\_retired:local\_hitm}) \\
& + 95 * \text{mem\_uncore\_retired:local\_hitm} \\
& + 250 * \text{mem\_uncore\_retired:local\_dram\_and\_remote\_cache\_hit} \\
& + 450 * \text{mem\_uncore\_retired:remote\_dram} \\
& + 450 * \text{mem\_uncore\_retired:remote\_hitm} \\
& + 250 * \text{mem\_uncore\_retired:other\_llc\_miss} \\
& + 7 * \text{dtlb\_load\_misses:stlb\_hit} \\
& + 7 * \text{dtlb\_load\_misses:walk\_completed} \\
& + 1 * \text{dtlb\_load\_misses:walk\_cycles} \\
& + 8 * \text{load\_block\_overlap\_store}
\end{aligned}
$$

Figure 6.7: Load Latency on Intel$^{\circledR}$ Westmere

Load Latency is calculated by multiplying the penalty of each event by the number of times this event occurred. For instance, on Intel$^{\circledR}$ Westmere, it is computed with the formula described in Figure 6.7. With that formula, it can be computed for an instruction, a function or a whole program.

The penalties have been calculated precisely by experiments on assembly.

Gooda uses Precise Event-Based Sampling (PEBS) for its Load Latency components to identify the assembly line as accurately as possible.

The converter simply gets the load latency value on each instruction and adds them on each inline stack. The value is multiplied by the multiplexing factor that Gooda used to have a normalized number.

6.7.4 Tests

The implementation of cache misses support has been tested with the Loop Fusion patch, see Section 7.

Chapter 7

Loop Fusion

The Loop Fusion optimization has been chosen as an attempt to use cache misses in a GCC pass. The details of this optimization are covered in Section 5.1. This chapter presents the implementation of this optimization in GCC and how it uses the Load Latency provided by Generic Optimization Data Analyzer (Gooda).

At the time of this writing GCC does not implement any Loop Fusion transformation. However, a patch providing this feature has been written[1]. To avoid reinventing the wheel, it has been decided to try this patch on the actual GCC trunk.

The patch has been adapted to the current version of GCC. Unfortunately, the loop merging algorithm was not working. Moreover, the tests verifying the legality of the transformation were not complete either and it has not been possible to find a more complete version of the patch. In view of these problems, it has been decided to write completely a new Loop Fusion pass for GCC.

7.1 The pass

Loop Fusion has been implemented as a new optimization pass, a Tree Static Single Assignment (SSA) pass (See Section 4.3). The Loop Fusion pass is executed as the first Tree SSA pass. The pass is automatically activated starting at optimization level `O2`.

Since August 2008, GCC has a new framework, Graphite, especially made for loop and memory optimizations [Sjodin2009]. This optimization could probably have

[1] `http://gcc.gnu.org/ml/gcc/2008-05/msg00184.html`

```
 1: $loops$ ← every innermost loops in the function
 2: for all pair $A$ and $B$ in $loops$ do
 3:    if $A$ is after $B$ in the instruction stream then
 4:       $Swap(A, B)$
 5:    end if
 6:    if $A$ and $B$ can be handled by the Loop Merging Algorithm then
 7:       if $A$ and $B$ are valid candidates for Loop Fusion then
 8:          if $A$ and $B$ are interesting candidates for Loop Fusion then
 9:             Fuse loops $A$ and $B$
10:             Apply a cleanup pass to the function
11:          end if
12:       end if
13:    end if
14: end for
```

Figure 7.1: Loop Fusion pass algorithm

been implemented in Graphite. Nonetheless, it has been decided to implement it as a Tree SSA pass. The reason of this choice is that Graphite is a complex framework that would have added to the difficulty of implementing the pass during the project. Moreover, there are more Tree SSA passes in GCC and they can be used as good examples on how to perform some recurring tasks in a pass implementation. Finally, Graphite does not seem to be used a lot and is not even always compiled into GCC on some system (e.g. Gentoo). Indeed, it needs some external dependencies rarely installed on a system by default: CLoog and ISL.

Although the transformation may look simple at first view, the implementation has been very challenging. Section 9.3 gives more details about the challenges encountered during the implementation.

The general algorithm is pretty straightforward. It is described in Figure 7.1.

This algorithm is divided in several parts:

1. Line 6: Test the limitations inherent to our strategy. See Section 7.2.
2. Line 7: Test the conditions for a loop fusion to be legal. See Section 7.3.
3. Line 8: Test that the loops are interesting candidates. See Section 7.4.
4. Line 9: Fuse two loops. See Section 7.5. The deletion of `B` will be automatically handled by the iteration over the loops provided by GCC.

5. Line 10: Make sure the Control Flow Graph (CFG) is clean. See Section 7.6.

After all possible candidate loops have been fused, several steps are necessary:

1. The virtual operands are marked for renaming. The SSA form contains special statements: Virtual USE and Virtual DEF. These statements indicate that a variable used as an operand is not directly a reference to another value (the variable has a partial or ambiguous reference). These virtual operands are hard to maintain directly, but they can be updated easily using a GCC function marking each of them for later renaming. They will be updated on the next SSA update (performed at Step 3).

2. The function is rewritten into Loop Closed SSA. Once changes are made, it is necessary to update the LCSSA form. This form makes updating the SSA form a bit easier than in normal SSA form. This could be done during fusion or even before, but this would introduce PHI nodes in the basic block between the loops. Considering the limitations of the merging strategy, this would prevent the newly created loop to be fused again with another sibling loop. Even if this transformation is not useful for this pass, it is necessary to perform it because the pass manager expects all the loop transformation passes to produce a valid LCSSA form.

3. The SSA form is updated again. This is necessary for the virtual operands to be renamed.

These steps are done only if at least one fusion has been performed.

During the pass, two metrics are collected using the event counters system provided by GCC:

- The number of loops fused.

- The number of basic blocks merged by the cleanup pass.

These statistics are used during the tests to verify the correctness of the transformations.

7.2 Limitations

The implemented merging strategy has some limitations on the loops that can be merged together. A and B can be merged together only if:

- They are both innermost loops. GCC offers a way to loop directly through all the innermost loops only. In practice, it would be possible to merge loops containing other loops by a more powerful fusion algorithm. It would also be necessary to ensure several properties on the sub loops.

- They have only two basic blocks each (one header basic block and one latch basic block). The number of basic blocks of the loop has to be equal to two and the header and latch basic blocks have to exist. These two blocks must be one after another in the instruction stream. The order of the two blocks is not important. This limitation can be improved by improving the fusion implementation. Several new cases should then be handled. It would largely increase the complexity of the fusion strategy.

- They have only one single exit. This is easily verified with the `single_exit` function. There are some cases where this limitation could be relaxed, but it is believed that only few loops with more than one exit could profit from this optimization.

- The latch basic blocks must be empty (no PHI nodes nor statements). This is necessary because the statements of the latch basic block are not transferred. In practice, if only the second loop has statements in the latch basic block or if the statements in the latch block are not dependent on the statements of the header, fusion remains possible. Removing this limitation would not complicate the fusion strategy too much.

- Their basic induction variables is of an integer type. It could be possible to merge loops with real types for instance as long as both loops have the same type. However, in practice, these kinds of loops would not profit much from Loop Fusion.

- The basic induction variable must have a corresponding SSA NAME that is not `NULL_TREE`. If this condition is not met, it is not possible to find the exact corresponding PHI node, and thus, not possible to rename the induction variables. This is a severe limitation as it prevents the fusion of some loops generated by the compiler.

- The basic induction variable must have a corresponding PHI node. This is necessary for the renaming of the induction variables. The renaming of variables is made using PHI nodes and their immediate use lists. Without PHI node, it would be necessary to make all the changes by hand without using the immediate list. That would complicate the fusion by a high margin.
- There is at most one basic block between `A` and `B`. The next basic block of the latch of `A` is the between block. The next basic block of the between block is the header of `B`. This block must meet several conditions:
  - It has no statements.
  - It can have PHI nodes as long as they are not used in the second loop. This can be verified by iterating through the list of immediate uses of the SSA Name of the PHI node and checking that no immediate use is in a basic block of the second loop.
  - It is the destination of the exit edge of the first loop and the predecessor of the entry of the second loop.
  - It has only one successor and one predecessor.
  - Note: There is a special case where the latch of the loop precedes its header. To be sure to have the right distance, it is necessary to get the last block of the first loop and the first block of the second loop. Then, the distance can be computed between these two blocks.
  - In practice, this condition could be relaxed. If there are several blocks with the correct properties, this should not block the fusion. Moreover, as long as the statements are independent from the first loop and the second loops statements are not dependent from their statements, it should be possible to move the statements in a block before the header of the first loop or after the second loop. Then, the fusion will be possible.

These limitations are only necessary because of the implemented strategy, they are not inherent to Loop Fusion in general.

Section 7.8.1 shows the impact of these limitations on the compilation of GCC.

7.3 Legality Conditions

It is necessary to verify some conditions before two loops can be merged to make sure that the semantic of the program will not be modified.

The conditions for which merging A and B is valid are the following:

1. A and B are sibling. For each loop, only the next loop is considered. As the loops are linked together in each loop nest, it is straightforward to take only siblings.
2. A and B are always executed together. The between block must have only one predecessor and one successor. A can only exit to the between block and the only predecessor of B is the between block.
3. A and B must iterate the same number of times. The loop analysis of GCC already computes the number of times a loop will iterate. A and B must have a precise estimation and both estimations should be the same. It could be possible to improve it to support loop iterating the same number of times, but with a number not-known at compile-time by recognizing some patterns.
4. The basic induction variable of A and B must be of the same type. Both types must be signed or both types must be unsigned. The type precision of the two types must be the same. The size of the type is not important as long as the precision and the signedness are the same.
5. A and B are independent. A does not need to fully performed before B.

Item 5 is the most complex to verify. For that, it is necessary to detect all the data references of the two loops, then to find the common dependences. A dependence can be of four types [Singhai1996]:

- A true-dependence: A writes to a location and B reads from it.
- An anti-dependence: A reads from the location and B writes to it.
- An output-dependence: A and B write to the location.
- An input-dependence: A and B read from the location.

The input dependence is the only one that can be violated. All the other dependences must not be reversed.

Figure 7.2 shows the legality test for two loops A and B.

For two data references to refer to the same base object (Line 5), they have to refer to a memory location common to both loops. For example, they can access the same

```
 1: refs_a ← all data references of A
 2: refs_b ← all data references of B
 3: for all ref_a in refs_a do
 4:    for all ref_b in refs_b do
 5:       if ref_a and ref_b refers to the same object then
 6:          if is_read(ref_a) and is_read(ref_b) then
 7:             It is an input dependence, always valid
 8:          else {Any other dependence type}
 9:             if not same_access_pattern(ref_a, ref_b) then
10:                A and B are not legal to fuse
11:             end if
12:          end if
13:       end if
14:    end for
15: end for
```

Figure 7.2: Loop Fusion Legality test

array, but with a different access variable. If two such data references are found, the access function must be the same (Line 9). Both access functions should use the basic induction variable and be equivalent.

When using the Data Dependency Analysis framework of GCC, there are several special cases to take into account:

1. The Data Dependency Analysis can fail. If it fails for some reason, there are not enough information to consider the legality of the fusion. The loops are not fused.

2. The Data Dependency Analysis retrieves an "unresolved" data reference. Sometimes the Data Dependency Analysis of GCC gives some incomplete data references (some fields are null). In this case, it has been decided to declare the loops non-mergeable to be safe.

3. The Data Dependency Analysis indicates a data dependence relation that it does not know about. When it happens, one of the data dependence relation is `chrec_dont_know`. In this case, the pass considers the whole analysis as failed and the loops are not fused.

7.4 Decision

Finally, not each pair of loops is an interesting candidate to Loop Fusion.

In this case, the decision is based on the following criteria:

- `A` and `B` are both small loops. The sum of their statements must be smaller than fifty. This is made to avoid creating a big loop body that could increase the number of Instruction Cache misses and thus, reduce the efficiency of the transformation.

- Large number of cache misses in `A` and `B`. The conditions $load\_latency(A) > 200$ and $load\_latency(B) > 200$ must be met.

- Number of misses in `A` $\approx$ Number of misses in `B`. There must be at least ten percent of difference between the two load latencies.

If these criteria are met, the pass prevails over the auto parallelization pass and can merge mixed loops (parallel with sequential).

Cache Misses information are available only when profile information is available and only when sampling has been used. To make the loop interesting when no profile information about cache misses is available, simple heuristics are used:

- `A` and `B` are both small loops. This is the same condition as above.

- `A` and `B` iterate a high number of times (more than 100) or their parent loop iterates a high number of times.

- `A` and `B` have a data reference on an array in common.

- `A` and `B` should be both parallel or both sequential loops. The iterations of a parallel loop can be executed in any order, making it a perfect candidate to be auto parallelized. If a sequential and a parallel loop are fused, the result will be a sequential loop that will not be parallelized, resulting in a possible loss of performance. The parallelizer of GCC is more advanced than this pass, so, when not enough information is available, the pass tries to give it the priority. If the auto parallelization feature is not activated, the status of the loops is not taken into account.

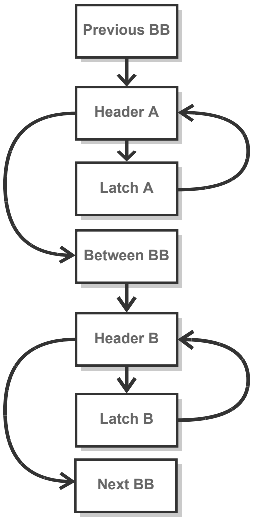


Figure 7.3: Loops Control Flow Graph

7.5 Loop Merging

Given the limitations of the algorithm (See Section 7.2), the candidate loops are in the form described by Figure 7.3.

A loop has two very important basic blocks:

- The header basic block. This block is the entry point of the loop.
- The latch basic block. This block contains a back edge to the header. Only one latch block is allowed in GCC representation of loops.

The algorithm to merge two loops is not straightforward. Considering two sibling loops (`A` and `B`) with each a header and a latch, the following tasks are necessary to merge `B` into `A`:

1. Replace the induction variables. The statements of `B` have to use the induction variable of `A`. First, it is necessary to find both induction variables. For that, the `GIMPLE_COND` statement that has to be present in each loop has to be found and one of its LHS or RHS operand has to be an SSA name. Once

the SSA name of the induction variable is known, the corresponding PHI node can be found as well. Once both PHI nodes are located, all the usages of the induction variable in B can be replaced by the induction variable of A.

2. Transfer the PHI nodes. All the PHI nodes of B not related to the Induction Variable have to be moved to A. As the arguments of the PHI nodes are directly related to the incoming edges of the basic block, it is not possible to move them, they have to be copied. For that, a new PHI node is created in A. The PHI argument definition and location have to be searched from the original edge. Then, they can be inserted into the new PHI node using the new edge. This has to be done for the edge from the previous block (the preheader) as well as for the edge coming from the latch.

3. Transfer the statements. All the statements of B are moved into A before the GIMPLE_COND statement. The GIMPLE_COND statement of B is not moved.

4. Delete loop B. The two basic blocks of B are removed from the loop and moved to the outer loop. Then, the loop B can be deleted. Finally, the basic block that was the latch of B is deleted.

5. Fix the edges. The edge from the old header of B becomes a fall through edge. The GIMPLE_COND statement is removed from this basic block.

6. Clean the basic blocks. This step is not mandatory to be a valid transformation. However, the previous steps create several empty basic blocks that can be removed. It is better for the following passes to have a CFG as clean as possible. There are two basic blocks that may be removed: the between block and the header of the second loop. The between block can be removed by redirecting the edge from A's loop header to the next block and then deleting the basic block. The between block is only removed if it is empty. The old loop header cannot be always directly removed at this point due to the possible use of PHI nodes referring to its edges in the next basic block. That is why a simple cleanup pass is made after each fusion

7. Update SSA. The SSA form is updated.

8. Update dominators. The dominators of each basic remaining blocks are recomputed.

7.6 Cleanup pass

To merge more than two loops together, it is necessary that the loop strategy does not leave empty basic blocks after the newly merged loop. On some cases, it is not possible to clean these empty basic blocks. To simplify the process of merging two loops, a cleanup pass is made after each fusion. This cleanup pass is simple, it tries to detect two basic blocks with these properties:

- Both are empty (no statement).
- Both have only one predecessor and one successor.
- One block fall through the other.

When two such basic blocks are found, they are merged using the CFG algorithm (the `merge_blocks` function). This cleanup pass makes it possible to fuse together several sibling loops.

7.7 Tests

As the pass is integrated in the optimization level `O2`, it is very important to test it thoroughly.

The pass is tested using two different means:

1. During the bootstrap. GCC is compiled in several phases. It is first compiled using the system compiler (stage1). Then, it is compiled using the GCC executable generated at the previous phase (stage2). The stage2 executable is used to compile again (stage3). Finally, the stage2 and stage3 are compared to see if GCC can compile itself. As the optimization pass is enabled for a default build of GCC, it will be compiled with the Loop Fusion pass. That helps a lot to find problems.
2. During the tests. GCC test suite includes more than 100,000 tests. Most of them are run with the optimization level `O2`. These tests will be compiled with the Loop Fusion optimization. Moreover, several tests specific to this pass have been integrated in the test suite to verify that loops are merged when it is adequate and not merged when it is not.

The tests have uncovered a lot of cases that were not taken into account during the development of the pass. The loops present in the test suite are very different. Different programming languages can produce different form of loops in the Intermediate Representation (IR). For instance, Fortran produced a large number of loops that have been useful to test the pass.

It has also been necessary to disable the pass for some tests. For instance, some tests about auto parallelization and vectorization do not work with loop fusion. Indeed, the tests are counting the number of parallelized and vectorized loops. This count is not the same after Loop Fusion removed some of the loops.

The pass has mainly been tested without Profile Guided Optimization (PGO). Indeed, the test suite and the compiler itself are compiled directly and it would have been hard to use PGO on them. For these reasons, the pass has been tested with PGO only on the samples that have been used to test the performances of the pass (See Section 7.8.2). The samples have been edited to iterate more times.

For these samples, the pass has been tested using some steps:

1. The sample is compiled using GCC (-O2) with the loop fusion pass disabled.
2. The executable generated at the previous step is profiled and the AFDO profile is generated.
3. The sample is compiled again with GCC (-O2 -fauto-profile) with the loop fusion pass. Debug messages are used to test if the loops are fused.

These tests have shown that the Load Latency of Gooda is a very small number. It has been necessary to adapt the previous decision function to adapt to those numbers. After that, the pass has been working with the numbers provided by PGO.

7.8 Results

This section presents the efforts made to verify the effectiveness, applicability and efficiency of this Loop Fusion pass.

7.8.1 Limitations

To see if this optimization could be effective, statistics have been gathered during the compilation of GCC itself compiled in `O2` with the Loop Fusion pass. Unfortunately,

Table 7.1: Loop Fusion Statistics for GCC

Reason	Count
Loop A is not mergeable	47644
Loop B is not mergeable	366
Loops are not mergeable together	573
Loops are not interesting to merge	0

Table 7.2: Detailed Loop Fusion Statistics for GCC

Type	Reason	Count
Condition	No estimate	11186
Condition	No precise estimate	88
Condition	The loops do not have the same estimation	363
Condition	The loops are not independent	3
Condition	Blocking dependence	2
Condition	Don't know data dependence relation	1
Limitation	The induction variable is not an integer	20
Limitation	The latch and the header blocks are not sibling	421
Limitation	The latch block is not empty	800
Limitation	No PHI node is corresponding to the induction variable	562
Limitation	The loop has more than one exit	71
Limitation	The SSA name for the induction variable is `NULL_TREE`	402
Limitation	The distance between the two loops is too high	69
Limitation	The loop has too many basic blocks	34460
Limitation	The between block contains too many statements	138

no loop has been fused in GCC. The loops in GCC are too complex to be fused regarding the limitations of this strategy.

Table 7.1 shows the reasons why each pair of loops has not been merged.

It is clear from the numbers that most of the loops are not mergeable by this pass. Most of the loops are not even considered in pair because they are not supported by the implementation. Table 7.2 shows the conditions blocking the process of merging loops.

The main factor blocking the loops from being merged in GCC is clearly the limitations of the implementation. The biggest limitation is the number of basic block in the loops. The implementation can get rid of this limitation, by improving the Loop

Fusion algorithm, . Then, the loops without an estimate. In that case, it is possible to recognize some patterns and verify that each loop iterates the same number of times without knowing this number. In practice, it is generally more current to have loops iterating an unknown number of times rather than loops with compile-time estimate. The other limiting factors are orders of magnitude smaller. However, removing the limitations on the PHI nodes for the Induction Variables could allow some loops to be fused. Allowing fusion of non-empty latch blocks could also bring some new opportunities of fusion. When only the second loop has statements in the latch block, the fusion remains possible.

7.8.2 Performances

The effectiveness of the optimization has been tested on several small use cases.

- `Read-Read`: Two loops read from the same array.
- `Write-Write`: Two loops write to the same array.
- `Write-Read`: The first loop writes to the array and the second reads from it.
- `Read-Write`: The first loop reads from the array and the second writes to it.
- `Crossed`: The first loop writes to A and reads B, the second reads A and writes to B.
- `Mixed`: Use of several arrays in both loops.
- `Copies`: 4 loops are performing array copies.
- `Dummy`: 2000 loops being 1000 copies of the two loops of `Write-Read`.

Each array is declared static. When they are read in the first loop, they are initialized in a first loop to prevent GCC from optimizing them. This first loop is never fused. The loops to be fused are inside an outer loop iterating 25 times. Each test is executed ten times and the mean of the execution times is computed.

```
#include <iostream>
static unsigned long A[40000000L];

int main(){
    for(unsigned long i = 0; i < 40000000L; ++i) A[i] = i;

    unsigned long a = 0, b = 0;
    for(int r = 0; r < 25; ++r){
        for(unsigned long i = 0; i < 40000000L; ++i) a += A[i];
        for(unsigned long j = 0; j < 40000000L; ++j) b += A[j];
    }

    std::cout << a << "," << b << std::endl;
    return 0;
}
```

Figure 7.4: Read-Read use case

For instance, Figure 7.4 shows the source code of the `Read-Read` use case.

The programs are compiled using a trunk version of GCC 4.8.0 with the options `-O2` and `-march=native`. All the tests were performed on a Lenovo Thinkpad W510 with an Intel® Core™ i7 Q820 processor, running at 1.73Ghz. The computer is running on Gentoo with the 3.6.0 Linux Kernel. Figure 7.5 shows the results obtained for each sample.

Of course, the sample for which Loop Fusion is the more profitable is `Dummy`. 2000 loops can be reduced to a few numbers of loops at the end for a large increase of performance. However, this sample is not a very realistic one.

Then, the sample profiting the most from the Loop Fusion is `Write/Write`, that is twice faster. It is logical because in this case, GCC can completely remove the first write to the array. The `Read/Read` example is also very interesting (1.7 times faster). Even if the other examples are less impressive, they are still from 1.3 to 1.5 times faster.

Figure 7.6 shows the number of cache misses before and after Loop Fusion.

Again, `Dummy` shows a huge decrease in the number of cache misses. In most cases, the number of cache misses is highly reduced (up to five times). The only sample that shows only small improvement is the `Write/Read` sample.

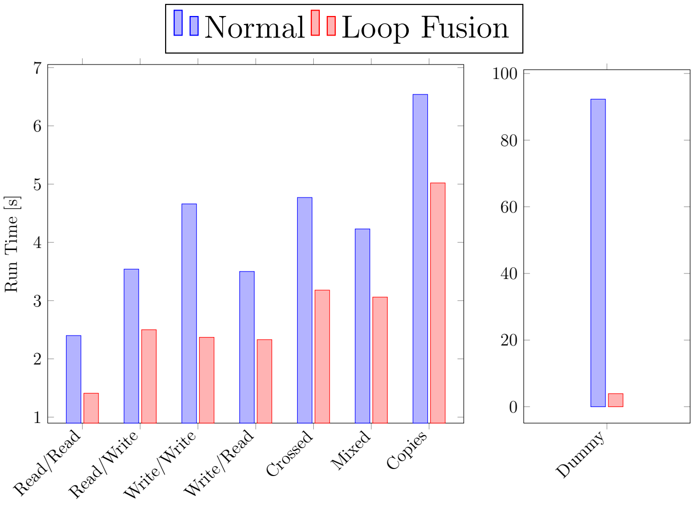


Figure 7.5: Loop Fusion Runtime Results

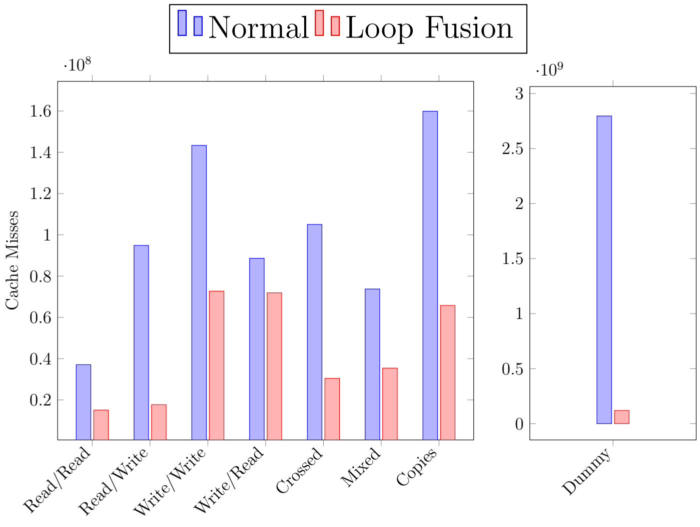


Figure 7.6: Loop Fusion Cache Misses Results

7.8.3 Efficiency

The efficiency of the Loop Fusion pass is important to incorporate it in GCC.

7.8.3.1 Complexity

The complexity analysis is done on a function level. During the compilation, the loop fusion pass will be executed for each function.

There are three components to evaluate:

1. The loop fusion itself
2. The cleanup pass
3. The analysis

The first two are less critical because they are only performed if two loops can be fused, which is rare in practice. Nevertheless, in the worst case, on a program with n loops, they will be executed $n - 1$ times, meaning that all the loops will be fused.

The Loop Fusion process is not very complex. Considering loops a and b, the Loop Fusion is performed in $\mathrm{O}\left(p_a + p_b + s_b\right)$, where p_x is the number of PHI nodes in loop x and s_x is the number of statements of loop x.

The Cleanup Pass tests if any two sibling basic blocks can be merged (the merge itself is in $\mathrm{O}\left(1\right)$ because only empty blocks are merged). It is performed in $\mathrm{O}\left(b\right)$ with b being the number of basic blocks of the current function.

The analysis is more critical because it will be performed on every pair of sibling loop, which can be a big number. In the worst case, the analysis is executed $\mathrm{O}\left(n\right)$, if all loops are siblings.

The critical part of the algorithm is the data dependence analysis. For two loops, it is performed in several steps:

1. All the data references of the loop are found.
2. All the data dependence relations between the data references are computed.
3. Verify that there are no blocking dependence between the two loops.

The first two steps are part of the data dependence analysis framework offered by GCC. They are made internally. It is invoked on each loop by the Loop Fusion Analysis. The third step is done by the Loop Fusion pass using the data of the first two steps.

The first step can be done in $\mathrm{O}\left(r\right)$, with r being the number of data references in the loop. Then, the second step is done in $\mathrm{O}\left(r^2\right)$. The third is performed in $\mathrm{O}\left(r_1 r_2\right)$. This gives an overall complexity of $\mathrm{O}\left(r^2\right)$.

The remaining part is made only on each loop separately. On a loop l, it can be done in $\mathrm{O}\left(s_l + p_l\right)$.

In the worst case, it can be safely assumed that the number of references is linear in the number of statements. With this assumption, the complexity is $\mathrm{O}\left(s_a s_b + p_a + p_b\right)$ for the analysis.

The complexity of the whole pass is dominated by the analysis part. The analysis is executed $n-1$ times in the worst case. Figure 7.7 shows the complexity analysis of the whole pass in the worst case.

$$\begin{aligned}
C_{cleanup} &= (n-1)b \\
C_{fusion} &= \sum_{i=0}^{n-1}(p_i + p_{i+1} + s_{i+1}) \\
C_{dependency} &= \sum_{i=0}^{n-1}(p_i + p_{i+1} + s_i s_{i+1}) = \sum_{i=0}^{n-1}(s_i s_{i+1}) \\
s &= \max_{i=0}^{n} s_i \\
p &= \max_{i=0}^{n} p_i \\
C_{dependency} &= (n-1)s^2 \\
C_{fusion} &= (n-1)(p+s) \\
C &= C_{dependency} + C_{cleanup} + C_{fusion} \\
&= (n-1)s^2 + (n-1)b + (n-1)(p+s) \\
&= (n-1)(s^2 + b + p + s) \\
&= ns^2
\end{aligned}$$

Figure 7.7: Loop Fusion Complexity Analysis

The whole pass complexity being $\mathrm{O}(ns^2)$, s being the maximum number of statements in a loop in the current function.

It is important to note that this complexity is for the worst case and that the worst case is only present if the function is composed only of loops and each loop is fused. It is believed that in practice, especially regarding the limitations of the algorithm, this analysis is much less costly.

7.8.3.2 Empirical Analysis

Section 7.8.3.1 showed that the complexity of this pass can be important. Nevertheless, it is believed that in practice the impact is not as important. To test this impact, the compilation times of several programs have been studied.

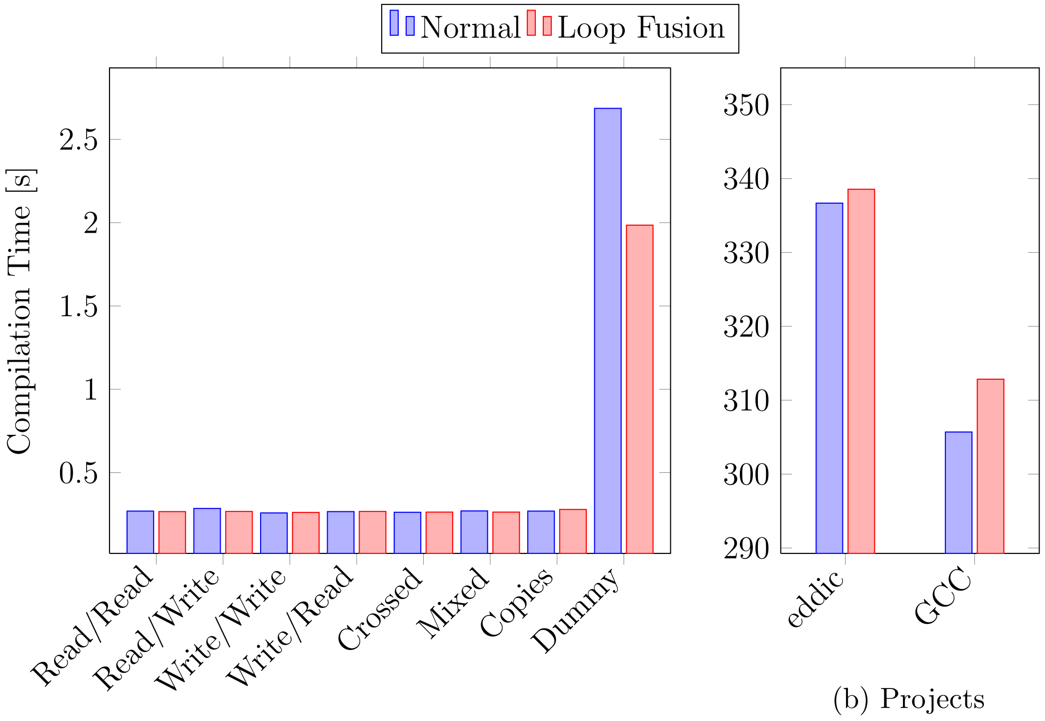
First, all the samples presented in Section 7.8.2 are compiled with GCC first with the pass deactivated and then with the default options. In all these samples, some loops will be fused. Each sample is compiled ten times and the mean of the ten compilation is taken. Figure 7.8a presents the compilation time obtained.

In the small samples, the difference in compilation time is negligible. Nevertheless, in the `Dummy` sample, the compilation is 20% faster with the patch. It is faster to merge loops than let the next passes analyze and optimize them.

To test the compilation time in real programs (where few or zero loops will be merged), two programs have been tested:

- GCC itself
- eddic, a small compiler written in C++

Figure 7.8b shows the results of these tests. This time, the compilation with the new pass is slower than the normal compilation, which is the normal result. Indeed, no loops have been merged in those two programs, but tests are performed for each loop whatsoever. However, the difference is not big: 2.3% for GCC and 0.5% for eddic.



(b) Projects

(a) Samples

Figure 7.8: Loop Fusion Compilation Time

Chapter 8

Use of Performance Events

At the beginning of this project, it was believed that knowing the number of Cache Misses in a compiler would be a great advantage. However, it turned out that this information was difficult to use efficiently. During this project, using it in Loop Transformations has been considered. Nevertheless, in all techniques that were considered, the information provided by the compiler was at least as efficient to use than the one provided by the Hardware Performance Events. The information was used in the Loop Fusion pass, but the decision could also have been implemented by statically predicting the memory behavior.

This chapter lists the main challenges related to the use of Performance Events in compiler optimizations. Some possible uses of events with a hypothetical Profile Guided Optimization (PGO) support are described in the last section.

8.1 Challenges

One problem comes from the support of PGO in modern compilers. The profiles are made for basic blocks and edges. Counters are aggregated for each basic block. For the counters related to the number of executions, this is not a problem as a basic block is especially made so that each of its instruction are executed together. But that is not the case for cache misses. Indeed, on a basic block basis, the number of cache misses become meaningless. A basic block can contain dozens of memory references, but only one counter is stored into the profile. With that property, it becomes really hard to define which instruction causes the cache misses. Many cache misses does not necessarily indicate a problem because some misses cannot be avoided.

Another problem is that for a basic block, it is generally easy to estimate the number of cache misses an instruction is responsible for. The compiler knows the layout of the data that are accessed. Moreover, most of the loop iteration layouts are also known at compile time. With this information, the compiler can predict the cache misses with a good accuracy. This prediction can be done on a memory reference level, an instruction possibly containing several references. It has been shown that most of the scientific loops are simple and regular loop nests [Ghosh1999] and that these loops can be statically analyzed with precise estimations of the cache misses. There also exists Interprocedural Algorithms for predicting the cache behavior and contents [Ferdinand1997].

Interpretation of the results is also an issue. How to know if the number of cache misses is high or normal is not an easy problem. The number of executions of a basic block can be easily translated in a frequency, but it is not as easy with cache misses.

The last problem that is important to consider when working with Hardware Performance Counters is the portability. There are differences between the events supported by different microarchitectures. There are also some differences in the precision at which a processor reports some events. It is possible than on a computer, sampling-based PGO would be very effective, but on another, the results could be much worse due to a poorer precision.

8.2 Possible uses

Having an instruction-profile for PGO would make cache misses and other counters more attractive and easier to use in an optimization technique.

The cost of such a profile should also be determined. It means that every instruction in the program will have an attached counter (or more, depending on the numbers of events). It will increase the memory usage as well as the cost of manipulating the data structures in a compiler.

For instance, the number of cache misses of each instruction could be used in Loop Fission. If a set of instructions is causing many data cache misses and they are independent from other instructions, the loop can be distributed in two smaller loops. The loop in where the cache misses did occur would now have a better spatial locality and the other instructions will not be fighting for the cache.

It could also be used in the decision of Loop Tiling. If there are not too many cache misses, tiling a loop would only slow down the loop nest.

The decision of Loop Fusion can also be improved. In Section 7, the decision is based on the number of cache misses in the basic block. Even if there are cache misses in both loops and if there are some common memory references, it does not always mean that it is interesting to fuse the loops. It could happen that the cache misses are occurring on instructions that are not manipulating these common references. Fusing the two loops could make the resulting loop slower because there will be more memory references in the fused loop. The decision should be based on the number of cache misses on the instruction sharing memory references.

Binary Layout could also be improved with this profile. Indeed, based on the statements manipulating the fields of structure, it could be possible to define a better layout by peeling or splitting the structure or simply by reordering the fields.

At the beginning of the project, Instruction Cache Misses have been left aside because of their poor precision. If this event was more precise, it could really be interesting for several optimizations. For instance, if there are already Instruction Cache misses in a loop, this loop should be optimized for size instead of being optimized for speed. For instance, such loops should never be unrolled further. The same can be achieved on a function level to optimize specific functions for size when there are already too much Instruction Cache misses.

Instruction Cache misses could also have been used for basic block ordering. When the instructions on the edge of two basic blocks are causing too many misses, several solutions exist. Tail duplication could be used to minimize instruction cache misses. The order of basic blocks can also be arranged to minimize them.

Function ordering could also compute a better function placement based on the instruction cache misses on the call instructions. This would require GCC to perform a full binary ordering of functions instead of just separating them in two sections.

Another event that has not been considered in detail and that could be interesting, if precise enough, is the number of branch misprediction. Indeed, too many branch mispredictions on a branch could be avoided by simply reversing the branch, making it more predictable. It could also indicate that some parts of the code should be reviewed. For example, a loop containing a single branch that is false half the time is the worst case for the Branch Prediction Unit. In some cases, this kind of loop can be rewritten in two loops avoiding most of the branch mispredictions.

All these events and probably other events as well could be used in a compiler if an instruction-profile was supported. If such profiles were available in compilers, it would open the door to interesting research on the use of these events.

Chapter 9

Challenges

This chapter lists the main challenges encountered during this project.

9.1 Performance Monitoring Events

Finding the list of events provided by a given microarchitecture has not been challenging. Indeed, this list is provided by Intel®. However, understanding the events has not always been easy.

Each event in the list is provided with a description. Nevertheless, this description is generally very short and not always clear. Moreover, these events are always referring to a lot of acronyms, not always easy to understand, e.g. DSB, RFO, PMH, SQ, etc. Some of the features of the microarchitecture are not very well-known, and thus it is hard to find information about what an event is referring to.

Furthermore, the information about the different events is somehow confusing. Some information available on a website may contradict the information of another source.

Understanding these events requires an advanced knowledge of the underlying microarchitecture.

Once the events are understood, it is also important to know how they are collected. Indeed, some events are collected with different precisions. Some of them are very precise, whereas some other events can be collected hundreds of instructions from the real location. This is very important to know before trying to use them to get performance speedups in applications.

9.2 GCC Optimization passes

The optimization passes of GCC are described in the GCC Internals documentation[1]. In this document, most of the passes seem to be clearly explained. Nevertheless, some descriptions are not complete regarding what the pass really performs.

It has been necessary to look at the source code to know exactly what a pass is doing. Most of the passes have a good source documentation, but some of them are very short and thus it is necessary to look directly at the source code of the pass.

After reading the sources, the GCC Internals documentation appears to be lacking a lot of passes and it describes passes not present anymore in GCC. Moreover, the order of the passes presented in the documentation does not reflect the order used in the source code.

Another difficult point has been to know which memory optimizations are performed on GCC. At the beginning, it was not really clear that most memory optimizations are loop and layout transformation techniques. Without knowing first the existing memory optimization techniques, it is not straightforward to find that does GCC to improve memory utilization.

9.3 GCC Development

Developing in GCC has been a real challenge. Not only GCC code base is really huge, it is also overly complex to understand.

It is not directly evident on how to develop in GCC. The configuration of the installation has to be changed to be able to debug GCC. All the passes have a corresponding dump file in which they can output their debug information. Nonetheless, it is not evident to enable them. The different data structures can also be dumped to the file or another flow, but again there are no centralized documentation on all this. Everything has to be looked for in the documentation or the source code.

Some data structures, functions and macros are documented in the GCC Internals documentation, but some of them are not up-to-date and sometimes it has been renamed or removed. The best documentation for finding a function is to look at the source code of other pass or to look at the data structures themselves. This is a time-consuming process, but most of the data structures, functions and macros are

[1] http://gcc.gnu.org/onlinedocs/gcc-4.7.1/gccint/

documented. An up-to-date documentation about developing for GCC would have helped a lot.

Adding a new pass is simple, but implementing it is really hard. Indeed, the analysis are run only once, so a pass has to keep all the analysis information up-to-date. This includes:

- The Loop Analysis information
- The Control Flow Graph (CFG)
- The Static Single Assignment (SSA) Form
- The Alias Analysis information
- The Dominator tree

Some parts are very complex and thus hard to maintain. Some functions are already taking care of updating some parts of the information, but rarely all of it. There are often several functions seeming to do the same thing, but performing radically different things. Moreover, some functions have a lot of unexpected side-effects.

For instance, the Loop Fusion pass is, in itself, very simple, it just consists in moving statements from one basic block to another. However, because it is necessary to maintain the whole analysis information, it becomes very complicated, especially because all loops are different and there are always exceptions to a simple case.

GCC contains powerful analysis utility, for instance the Data Dependence Analysis. The problem with those utilities is that it not always trivial to use them. Moreover, they are generally low-level. For instance, it is possible to find all the data dependencies of a basic block, but it is not possible to test if two basic blocks are independent, which is a common operation for Data Dependence Analysis.

The most challenging task is to maintain the SSA form, especially the PHI nodes. The PHI nodes are indicating which values a variable can hold at the beginning of the basic block in a function. These nodes are crucial to generate the real code after the program left the SSA form. It is very complex to update them and a lot of operations (redirecting edges for instance) seem to make them invalid. During the development of the Loop Fusion pass, it has been the hardest part to deal with.

The order of the operations is very important to not damage the intermediate representations. For instance, it is not possible to change the edges inside a loop before removing the loops from the program. It is necessary to remove the loop and then make changes to the edges. In the same way, it is not possible to cancel a loop tree after a basic block of the loop has been removed. And, it is not possible to remove a basic block that is not in a loop.

A good point when developing in GCC is that there are several functions to verify the correctness of the Intermediate Representation. There are functions to verify the validity of the loop structure, the GIMPLE CFG and the SSA form. However, when some things are really messed up, these functions simply fail (for instance with a Segmentation Fault error) instead of indicating the source of the error. Nevertheless, these functions have been of great help.

GCC contains a very high number of tests (more than hundred thousand). When the change is made to something activated at a given optimization level (`O2` in the case of this project), many tests are executed using the change. That is very good to ensure that the change is stable and safe. On the other hand, executing the whole test suite can take hours on a desktop computer. For this project, it has been necessary to use a high performance server to run the tests so that they finish in reasonable time to debug quickly the code. Moreover, it seems that the test suite does never pass entirely the tests. For that, it is necessary to run the tests once without the patch. Then, again with the patch and compare the results of the two runs.

An important thing to note is that nothing can be assumed when compiling. Everything has to be checked. For instance, every test case in C and C++ that has been tested had the basic block edges in the same order. Nevertheless, the Fortran test cases had another order. It is very important to program defensively when creating a new pass for GCC.

9.4 Gooda

Generic Optimization Data Analyzer (Gooda) is not a very convenient tool to use for several reasons:

- It needs an external library: libpfm.
- It works only with a patched version of the `perf` tools. As the `perf` tools are included in the kernel, a patched version of the kernel is necessary. The patch is necessary for the `perf` tools to use libpfm to get the correct events.
- It works only on some Intel® microarchitectures, namely Intel® Westmere, Intel® Sandy Bridge and Intel® Ivy Bridge.

For these reasons, all the tests involving Gooda have been done in a separate machine, a server of the LBNL.

Even when these conditions are met, Gooda is not always very easy to use at the first view. Several scripts are available and it is necessary to know the specific model of the processor to choose the correct script. Moreover, there are several scripts available for each platform and it is not very clear which one has to be used.

The goal of Gooda is to abstract the complexity of performance analysis on Intel® processors. This goal is not completely achieved. Indeed, some of the events of the spreadsheets are hard to understand and are not documented anywhere.

Even if the source code is available, it does not help understand how the tool works. Indeed, the code is full of debug information (defined by macros), parts of commented code, hundreds of line function, poorly formatted code, etc.

At the beginning of the project, it seemed that Gooda used Last Branch Record (LBR) for precise profiling. Nonetheless, it turned out that only one script provided this feature. Moreover, the use of LBR and Gooda together was not very stable. It caused infinite loops and imprecise reporting of Gooda. These problems have then been fixed, but delayed a lot the testing.

9.5 GCOV

The GCOV library is not really made to be integrated in another application, especially one in C++.

It is necessary to include the GCOV files contained in the GCC source and build folders in the application. The source file `gcov-io.c` has to be included as well. Moreover, the header `gcov-io.h` contains several definitions and thus it is not possible to include it several times in the application. All these files are controlled by several macros. These macros determine the mode in which the file has to be compiled (GCOV library, GCC or the GCOV tool). Each mode has access to different functions. It is not clear which macro to use when including the GCOV files. The GCC build directory is also necessary because the file with the GCOV version is generated during compilation of GCC.

Furthermore, the files are not self-sufficient, it is necessary to include other headers from GCC without any documentation on which headers to include. It can work differently with different combination of included headers.

For all these reasons, the GCOV library has not been used in this project. The functions to write to and read from the GCOV format have been rewritten in C++.

9.6 SPEC

The SPEC benchmarks are very complete and contains several tools. On the other hand, they are hard to configure to achieve complex tasks. They are not really flexible. Moreover, the documentation is somehow cryptic.

The benchmarks take a very long time to build and complete. In average, about five minutes are necessary to build a benchmark and more than ten minutes to run it. As each benchmark needs to be run at least three times, it takes one hour to generate the reference results for one benchmark. This needs to be multiplied by the number of configurations tested (four for this project) and by the number of benchmarks.

A feature that seems as essential as comparing the results of two different configurations on the same benchmark is not provided. It has been necessary to write several scripts to automate the runs, parse the results, aggregate them and finally compare the different versions.

The tools do not provide any support for parallel compilation or parallel execution by default. Everything has to be done by other tools to run benchmarks in parallel. The only thing that is done in parallel is the generation of the build and run directories, which is not really important compared to the time necessary to run the benchmarks themselves. As each benchmark easily consumes about 2GB of memory, running them in parallel needs a solid configuration.

Regarding Profile Guided Optimization (PGO), there is support to make compilation in several passes including a training pass with a specific data set. However, some of the benchmarks have several workloads and also several training workloads. If a benchmark has two workloads, the program will be trained two times, once with each training workload, and then run two times, once with each reference workload. In the case of this project, the toolchain does not aggregate previous profiles, so the first training is useless. Furthermore, the first run is made with a training workload that is not necessarily relevant. There is no way to change this behavior simply. Finally, it has been necessary to create custom versions of some of the benchmark to make sure that only one data set were used.

The benchmarks contained in the suite have proved very useful and producing stable results. On the other hand, the SPEC tooling has proved quite disappointing.

Chapter 10

Conclusion

Software Optimization based on Performance Monitoring Events is a complicated science. Tools like Generic Optimization Data Analyzer (Gooda) can help users by abstracting some of the complexity. However, even with such tool, it is still necessary to understand several complex concepts. I strongly believe that the use of Hardware Events can improve the quality of software as well as simplify the optimization process. Nevertheless, it would be necessary to improve the existing tools to reach a higher number of users. One way to use Hardware Performance Counters without requiring a large knowledge of the events is to automate their use in compilers.

The goal of this project was to improve the process of Profile Guided Optimization (PGO). For that, Hardware Performance Events have been studied and ways to integrate them in a compiler have been considered.

To achieve these goals, a new toolchain has been developed. The profiling is performed with `perf` and Gooda. Then, an application developed during the project is converting the Gooda spreadsheets into an instruction-profile. GCC is finally using this profile by using the AutoFDO patch, developed by Google. The complete toolchain is made of open source projects. It is important in order to make it sustainable and open to future contributions. This model has proved effective with the Linux kernel itself.

The efficiency of the sampling-based PGO has been compared to the traditional instrumentation-based approach. The results obtained with the toolchain are very satisfying. On average, the toolchain achieves 83 percent of instrumentation-based PGO gains. On C++ benchmarks only, it achieves 93 percent of the gains (AutoFDO has been especially optimized for C++ programs). The average overhead is only 1.06

percent while instrumentation incurs 16 percent overhead on average. I think that Sampling-based PGO is a really promising technique that should be used more and more in the future. It is even possible that this technique becomes more effective than instrumentation-based version by fine tuning the profiles and by using several events in the profile to tune other optimization techniques.

In a second phase, Cache Misses have been integrated in the profile generated by the toolchain. A new optimization, Loop Fusion, has been implemented for GCC in order to use this new profile information. This pass has been successfully tested on GCC. The transformation proved very efficient on small samples, increasing performance from 20 to 100 percent. However, on complete applications, no performance improvement has been observed.

This project did not present any revolutionary use of cache misses in a compiler. Indeed, it was more difficult to use them in modern optimizations that it was believed at the beginning of the project. Nevertheless, it is still believed that the addition of several new information in the profile coming from the hardware counters can be an advantage for a compiler. Having the information available can lead other projects to use it. Moreover, now that the infrastructure is implemented, it becomes easy to add support for new counters.

10.1 Future work

This section lists some future tasks that could be done to improve or continue the work performed during this project.

10.1.1 Sampling-based PGO

Several tasks are necessary to make the toolchain available to as much user as possible. The AutoFDO implementation is currently only compatible with the Google branch of GCC. The AFDO patch should be adapted for GCC trunk. Moreover, Gooda supports only Intel® Westmere, Sandy Bridge and Ivy Bridge. Support for PowerPC and ARM is currently under development. Adding support for new processors in Gooda would be useful to facilitate its adoption. Finally, AutoFDO is especially tuned for C++ programs. In the future, it would be very interesting to make it more powerful for other programming languages, for instance Fortran.

This project adds support for sampling-based PGO and cache misses information.

Adding new counters (e.g. branch misprediction) is very straightforward in the implementation. However, adding a new counter in GCC implies adding a new field in the `basic_block` structure. This structure is used a lot in GCC and adding several new fields will increase the memory usage of GCC that is already problematic. In order to add these new fields and not increase memory usage too much when they are not used, it would be necessary to use another data structure, with small memory overhead when it is empty, to store the profile.

Several techniques could be used to improve the accuracy of the generated profile. The first technique would be to use a Minimum Control Flow algorithm [Chen2010] to smooth the number of executions of the basic block. Another technique would be to correlate the information provided by PEBS events and the LBR event in order. As Gooda also performs a call graph analysis, this script could be used to compute much more accurately the number of executions of a function.

Instrumentation-based PGO has still an advantage over this solution with handling value profiles. This kind of profile is not yet supported by AutoFDO. However, it has already been implemented successfully with sampling profiles by Dehao Chen et al. [Chen2010], so it could be done again in the present toolchain.

The time necessary to generate the profile is highly depending on the performances of `addr2line` and `objdump`. The converter generates up to three calls to these programs per ELF file. This could be improved by grouping together the two `addr2line` invocations. This would eventually improve the running time.

The converter should also be tested on more architectures as `perf` and Gooda are very dependent on the microarchitecture. At the present time, it has only been tested on Intel® Sandy Bridge.

For now, it is not possible to use both cache misses and Last Branch Record (LBR) at the same time because Gooda cannot compute Load Latency in LBR mode. The problem is that `perf` cannot activate LBR on only one event. When LBR is activated on one event, all the other events are using it as well. A patch has been written by the Gooda team to fix this problem, but the patch has not yet been accepted into the Linux Kernel. Once `perf` and Gooda are updated, it will be easy to add cache misses support to LBR as well. Especially because the LBR version has a much lower overhead than Cycle Accounting and provides better accuracy.

In multithreaded application, it often happens that multiple threads are accessing the same cache line, with one of them writing to it. In that case, the whole cache line will be set as dirty and will need to be retrieved from the main memory. If the threads are accessing the same data, it is not a problem, but it often happens that the threads are accessing different data in the same cache line. This situation is called False Sharing [Tolubaeva2012, Jeremiassen1995]. In that case, it can be highly interesting to force the two data to be on different cache lines. If the compilers had access to a per-thread profile, it could perhaps prevent this from happening. There are other cases where a per-thread profile could certainly be profitable for compilers.

Moreover, it has also been found that using some events in a basic block profile was not accurate enough to use them. Having an implementation of an instruction profile would be very interesting to test new events and their use in compiler optimization.

10.1.2 Loop Fusion

The Loop Fusion patch can definitely be improved. At this time, due to the limitations of the implementation, only few loops can be fused. It is possible to improve the algorithm to support more types of loops. At this time, the pass merges only loops with compile-time estimates. However, in practice, most of the loops do not have compile-time iteration estimate. Under some conditions, it is possible to merge loops with the same iteration pattern even if the number of iterations is not known at compile-time. It could also be especially interesting to support merging loops that have more than two basic blocks. Loops with statements and PHI nodes in the latch basic blocks could be merged too by improving the Fusion algorithm.

The performed data dependence legality test is correct, but is very conservative. Some of the tests can be relaxed. The current test verifies that the access functions are the same. However, in some cases, it is enough to verify that the dependency direction vector is either (=) or (<). For instance, if an iteration needs a value being computed in the past, it should not prevent the loop from being fused because the data will be available after the previous iteration.

Some methods can be applied before and after loop fusion to make it more efficient [Manjikian1997]. It is possible to fuse more loops by shifting iteration spaces. It is also possible to use peeling after two loops have been merged in order to retain loop parallelism during Loop Fusion. As Loop Fusion can be profitable by enabling Loop Permutation [Carr1994], it could be interesting to prioritize fusion of loops that could make possible a new permutation.

Finally, the performances of the pass can certainly be improved. The tests that are performed for the legality test should be performed in such an order that invalid loops are pruned as early as possible. Moreover, the Data Dependence Analysis is performed twice, once for the validity analysis and once to test if the loops are interesting to merge. Doing it once should not make the implementation much more complicated and should improve the performances.

Once some improvements have been made on the patch, it will be necessary to propose it to the GCC developers to see if they find this new optimization interesting and want to integrate it. This will eventually be done after this project with hope to have it released in GCC 4.9.

10.2 What I learned

This project was a great opportunity to dig into GCC. I learned a lot about the GNU Compiler Collection internals. Due to the lack of documentation or the outdated resources, the best documentation for developing with GCC is to read the source code of other optimization passes. I studied several optimization passes and other internal source code to learn how to make things right in GCC.

Not only GCC is much clearer to me than before, now, but I obtained new knowledge in compilers in general. I studied in detail several optimization techniques that I did not know before this project. I think that I have a good understanding of PGO in general and how to use profile data to improve existing optimizations.

It also has been a good occasion to learn about the Intel® microarchitecture, especially Intel® Ivy Bridge. The way instructions are handled in this microarchitecture is much clearer to me than before. I also understand pretty well how memory caches work in a processor. More than the microarchitecture, I also acquired knowledge in Hardware Performance Monitoring. I am aware of the type of events that are available, their richness as well as their limitations. It has also been an opportunity to study how to use them in existing compiler optimizations.

I also learned how to use the capabilities of Gooda and `perf` to perform Cycle Accounting of an application. I am now aware of several performance factors that were obscure before: for instance, the importance of the memory hierarchy, the importance of the microarchitecture or false sharing.

Finally, as Gooda requires a patched kernel and the benchmarking server has been completely installed from scratch, I also renewed my knowledge in Linux server and working with the Linux kernel. Several configuration changes have been necessary to stabilize the results on the server to have correct results when benchmarking on the SPEC suite. It will help me manage complete benchmarks in the future.

Appendix A

Content of the archive

In appendices of this document, you will find an archive containing the following elements:

- This document in PDF: report.pdf
- The logbook in PDF: logbook.pdf
- `report`: The LaTeX sources of the report
- `gcc-loop-fusion`: The sources of the GCC Loop Fusion patch as well as the GCC/AFDO patch for cache misses
- `gooda-to-afdo-converter`: The sources of the converter from Generic Optimization Data Analyzer (Gooda) to the AFDO format.
- `profiles`: The AFDO profiles that have been provided by Dehao Chen (Google)

The implementation parts of this project are also available on their respective repositories:

- The loop fusion patch: `https://github.com/wichtounet/gcc-loop-fusion`
- The Gooda to AFDO converter: `https://github.com/wichtounet/gooda-to-afdo-converter`

Appendix B

Installation

This chapter presents the installation of the different implementation of this project.

All the scripts are using as many cores as possible. If it not a desirable behavior, the -j option of make commands has to be removed or modified.

The converter has several dependencies in order to be built and installed. It is necessary to have Git to get the sources and CMake to build them. It needs at least GCC 4.7 and Boost 1.44 to build. The converter can be installed from the Git repository. Then, it can be built with the commands shown in Figure B.1

```
git clone git://github.com/wichtounet/gooda-to-afdo-converter.git
    converter_src
mkdir converter_build
cd converter_build
cmake ../converter_src/
cores=`cat /proc/cpuinfo | grep processor | wc -l`
threads=$(($cores+1))
make -j$threads
```

Figure B.1: Install the converter

To have a working installation of GCC with Sampling Profiling, it is necessary to install the AFDO patch on top of GCC. At the time of this writing, the patch is specifically made for the Google branch of GCC. The patch has already been applied to the GCC 4.7 Google branch. It is also necessary to install the patch bringing cache misses support to GCC. The Loop Fusion patch is not mandatory. This installation can take a long time depending on the computer, up to several hours on a single core processor. Figure B.2 shows the commands that are necessary to build it. The sources will be installed in the `gcc_src` directory, the build files in `gcc_build` and the installation files in `gcc_install`. The `gcc_src` and `gcc_build` directory can be removed if not interesting.

```
mkdir gcc_build
mkdir gcc_install
svn co svn://gcc.gnu.org/svn/gcc/branches/google/gcc-4_7 gcc_src
cd gcc_src
patch -p0 -i gcc_misses.diff
patch -p0 -i loop_fusion.diff
cd ../gcc_build/
../gcc_src/configure --prefix=`readlink ../gcc_install/`
cores=`cat /proc/cpuinfo | grep processor | wc -l`
threads=$(($cores+1))
make -j$threads
make install
```

Figure B.2: Install AFDO

Figure B.3 shows how the converter can be used to profile an application and generate an AFDO profile. Generic Optimization Data Analyzer (Gooda), `perf`, `objdump` and `addr2line` must be installed before the converter can be used.

```
cd your_application_folder/
../converter_build/bin/converter --profile --afdo your_application
   your_options
../gcc_install/bin/g++ -fauto-profile gcc_options your_application.c
```

Figure B.3: Profile the application and compile it with AFDO

The Loop Fusion pass can also be directly applied to GCC source code, without AFDO. Figure B.4 describes how to achieve that.

```
mkdir gcc_build
mkdir gcc_install
svn co svn://gcc.gnu.org/svn/gcc/trunk gcc_src
cd gcc_src
patch -p0 -i loop_fusion_no_afdo.diff
cd ../gcc_build/
../gcc_src/configure --prefix=`readlink../gcc_install/`
cores=`cat /proc/cpuinfo | grep processor | wc -l`
threads=$(($cores+1))
make -j$threads
make install
```

Figure B.4: Install the Loop Fusion patch

Appendix C

Performance Events for Intel® Ivy Bridge

This chapter presents the list of Performance Monitoring Events supported by the third Generation Intel® Core™ Processors. This generation is based on the Intel® Ivy Bridge microarchitecture.

These tables have been extracted from [Intel2012].

The events are separated in two different categories:

1. Architectural Performance Events (See Table C.1). These events have a behavior that is consistent across processor implementations.
2. Non-Architectural Performance Events (See Table C.2). These events are processor-specific.

Both categories supports sampling and counting as well and are accessed the same way.

Table C.1: Architectural Performance Events for Ivy Bridge microarchitecture

Event Mask Mnenonic	**Description**
UnHalted Core Cycles	Unhalted core cycles
UnHalted Reference Cycles	Unhalted reference cycles
Instruction Retired	Instruction retired
Last Level Cache (LLC) Reference	LLC references
LLC Misses	LLC misses

Table C.1 Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
Branch Instruction	Retired Branch instruction at retirement
Branch Misses Retired	Mispredicted Branch Instruction at retirement

Table C.2: Non-Architectural Performance Events for Ivy Bridge microarchitecture

Event Mask Mnenonic	**Description**
`LD_BLOCKS.STORE_FORWARD`	loads blocked by overlapping with store buffer that cannot be forwarded.
`MSSALIGN_MEM_REF.LOADS`	Speculative cache-line split load uops dispatched to L1D.
`MSSALIGN_MEM_REF.STORES`	Speculative cache-line split Store-address uops dispatched to L1D.
`LD_BLOCKS_PARTIAL.ADDRESS_ALIAS`	False dependencies in MOB due to partial compare on address.
`DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK`	Misses in all TLB levels that cause a page walk of any page size from demand loads.
`DTLB_LOAD_MISSES.WALK_COMPLETED`	Misses in all TLB levels that caused page walk completed of any size by demand loads.
`DTLB_LOAD_MISSES.WALK_DURATION`	Cycle PMH is busy with a walk due to demand loads.
`UOPS_ISSUED.ANY`	Increments each cycle the # of Uops issued by the RAT to RS.
`UOPS_ISSUED.FLAGS_MERGE`	Number of flags-merge uops allocated. Such uops adds delay.
`UOPS_ISSUED.SLOW_LEA`	Number of slow LEA or similar uops allocated. Such uop has 3 sources (e.g. 2 sources + immediate) regardless if as a result of LEA instruction or not.
`UOPS_ISSUED.SiNGLE_MUL`	Number of multiply packed/scalar single precision uops allocated.
`ARITH.FPU_DIV_ACTIVE`	Cycles that the divider is active, includes INT and FP.
`L2_RQSTS.DEMAND_DATA_RD_HIT`	Demand Data Read requests that hit L2 cache
`L2_RQSTS.ALL_DEMAND_DATA_RD`	Counts any demand and L1 HW prefetch data load requests to L2.
`L2_RQSTS.RFO_HITS`	Counts the number of store RFO requests that hit the L2 cache.
`L2_RQSTS.RFO_MISS`	Counts the number of store RFO requests that miss the L2 cache.
`L2_RQSTS.ALL_RFO`	Counts all L2 store RFO requests.
`L2_RQSTS.CODE_RD_HIT`	Number of instruction fetches that hit the L2 cache.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.
L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.
L2_RQSTS.PF_HIT	Counts all L2 HW prefetcher requests that hit L2.
L2_RQSTS.PF_MISS	Counts all L2 HW prefetcher requests that missed L2.
L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.
L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines
L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state
L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state
L2_L1D_WB_RQSTS.MISS	Not rejected writebacks that missed LLC.
L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.
L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.
L2_L1D_WB_RQSTS.ALL	Not rejected writebacks from L1D to L2 cache lines in any state.
LONGEST_LAT_CACHE. REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.
LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.
CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.
CPU_CLK_THREAD_UNHALTED. REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.
L1D_PEND_SS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmaks = 1 and Edge =1 to count occurrences.
DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes an page walk of any page size (4K/2M/4M/1G).
DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).
DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.
DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks
LOAD_HIT_PRE.SW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.
L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.
MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer Move Elimination candidate uops that were not eliminated.
MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD Move Elimination candidate uops that were not eliminated.
MOVE_ELIMINATION.INT_ELIMINATED	Number of integer Move Elimination candidate uops that were eliminated.
MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD Move Elimination candidate uops that were eliminated.
CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.
CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.
RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.
DTLB_LOAD_MISSES.STLB_HIT	Counts load operations that missed 1st level DTLB but hit the 2nd level.
OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore.
OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding Demand Code Read transactions in SQ to uncore.
OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore.
OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore.
LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.
LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.
IDQ.EMPTY	Counts cycles the IDQ is empty.
IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path.
IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path.
IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE.
IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE.
IDQ.ALL_DSB_CYCLES_ANY_U	OPS Counts cycles DSB is delivered at least one uops.
IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops.
IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uops.
IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops.
IDQ.MITE_ALL_UOPS	# of uops delivered to IDQ from any path.
ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.
ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks
ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks
ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.
ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.
ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.
ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.
BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.
BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.
BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls nor returns.
BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.
BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non call branch, executed.
BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.
BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.
BR_INST_EXEC.TAKEN	Qualify taken near branches executed.
BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).
BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.
BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls nor returns.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.
BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non call branch, executed.
BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.
BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.
BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed.
BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).
IDQ_UOPS_NOT_DELIVERED.CORE	Count number of non-delivered uops to RAT per thread.
UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.
UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.
UOPS_DISPATCHED_PORT.PORT_2_LD	Cycles which a load uop is dispatched on port 2.
UOPS_DISPATCHED_PORT.PORT_2_STA	Cycles which a store address uop is dispatched on port 2.
UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.
UOPS_DISPATCHED_PORT.PORT_3_LD	Cycles which a load uop is dispatched on port 3.
UOPS_DISPATCHED_PORT.PORT_3_STA	Cycles which a store address uop is dispatched on port 3.
UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.
UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.
UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.
RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.
RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.
RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).
RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.
CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
CYCLE_ACTIVITY.CYCLES_LDM_PENDING	Cycles with pending memory loads.
CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads.
CYCLE_ACTIVITY.CYCLES_NO_EXECUTE	Cycles of dispatch stalls.
DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.
DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.
DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered >3 DSB lines.
ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.
OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.
OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.
OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM
OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).
UOPS_EXECUTED.THREAD	Counts total number of uops to be executed perthread each cycle.
UOPS_EXECUTED.CORE	Counts total number of uops to be executed percore each cycle.
OFFCORE_RESPONSE_0	see Section 18.8.5, “Off-core Response Performance Monitoring”.
OFFCORE_RESPONSE_1	See Section 18.8.5, “Off-core Response Performance Monitoring”.
TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.
TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.
INST_RETIRED.ANY_P	Number of instructions at retirement.
INST_RETIRED.ALL	Precise instruction retired event with HW to reduce effect of Precise Event-Based Sampling (PEBS) shadow in IP distribution.
OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.
OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.
UOPS_RETIRED.ALL	Counts the number of micro-ops retired.
UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.
MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due tomemory order conflicts.
MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.
MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.
BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.
BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired. Supports PEBS
BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.
BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.
BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.
BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.
BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.
BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.
BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.
BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.
BR_MISP_RETIRED.NEAR_CALL	Direct and indirect mispredicted near call instructions retired.
BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.
BR_MISP_RETIRED.NOT_TAKEN	Mispredicted not taken branch instructions retired.
BR_MISP_RETIRED.TAKEN	Mispredicted taken branch instructions retired.
FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to Output values.
FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.
FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to Output values.
FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
FP_ASSIST.ANY	Cycles with any input/output SSE or FP assists.
ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.
MEM_TRANS_RETIRED.LOAD_LATENCY	Sample loads with specified latency threshold.
MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.
MEM_UOPS_RETIRED.LOADS	Qualify retired memory uops that are loads.
MEM_UOPS_RETIRED.STORES	Qualify retired memory uops that are stores.
MEM_UOPS_RETIRED.STLB_MISS	Qualify retired memory uops with STLB miss.
MEM_UOPS_RETIRED.LOCK	Qualify retired memory uops with lock.
MEM_UOPS_RETIRED.SPLIT	Qualify retired memory uops with line split.
MEM_UOPS_RETIRED.ALL	Qualify any retired memory uops.
MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.
MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.
MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops with LLC cache hits as datasources.
MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops which data sources were data missed LLC (excluding unknown data source).
MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were LLC hit and cross-core snoop missed in on-pkg core cache.
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were LLC and cross-core snoop hits in on-pkg core cache.
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared LLC.
MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in LLC without snoops required.
MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops which data sources missed LLC but serviced from local dram.
BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.
L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache.
L2_TRANS.RFO	RFO requests that access L2 cache.

Table C.2 Non-Architectural Performance Events for Ivy Bridge microarchitecture (Contd.)

Event Mask Mnenonic	**Description**
`L2_TRANS.CODE_RD`	L2 cache accesses when fetching instructions.
`L2_TRANS.ALL_PF`	Any MLC or LLC HW prefetch accessing L2, including rejects.
`L2_TRANS.L1D_WB`	L1D writebacks that access L2 cache.
`L2_TRANS.L2_FILL`	L2 fill requests that access L2 cache.
`L2_TRANS.L2_WB`	L2 writebacks that access L2 cache.
`L2_TRANS.ALL_REQUESTS`	Transactions accessing L2 pipe.
`L2_LINES_IN.I`	L2 cache lines in I state filling L2.
`L2_LINES_IN.S`	L2 cache lines in S state filling L2.
`L2_LINES_IN.E`	L2 cache lines in E state filling L2.
`L2_LINES_IN.ALL`	L2 cache lines filling L2.
`L2_LINES_OUT.DEMAND_CLEAN`	Clean L2 cache lines evicted by demand.
`L2_LINES_OUT.DEMAND_DIRTY`	Dirty L2 cache lines evicted by demand.
`L2_LINES_OUT.PF_CLEAN`	Clean L2 cache lines evicted by the MLC prefetcher.
`L2_LINES_OUT.PF_DIRTY`	Dirty L2 cache lines evicted by the MLC prefetcher.
`L2_LINES_OUT.DIRTY_ALL`	Dirty L2 cache lines filling the L2.

Appendix D

GCC Passes

- All lowering passes
  - Warn about unused result
  - Diagnose OpenMP and TM blocks violations
  - mudflap pass 1
  - Lower OpenMP and TM constructs
  - Lower cf (switch to low GIMPLE)
  - Refactor exception handling code
  - Lower exception handling code
  - Build control flow graph
  - Warn about function return
  - Build call graph edges
- All small IPA passes
  - Free language specific information
  - Mark visibility of functions and variables
  - Early local passes
    - Fixup Control Flow Graph (CFG)
    - Initialize data structures
    - Expand OpenMP constructs
    - Find all reference vars
    - Convert to Static Single Assignment (SSA) form
    - Lower vector operations to scalar operations
    - Warn about uninitialized variables
    - Rebuild the call graph edges
    - Inline analysis

    - Early inliner (local)
    - All early optimization
      - Remove call graph callees edge
      - Rename SSA copies
      - Conditional constant propagation
      - Forward propagation of single-use variables
      - Rebuild alias analysis
      - Early Intraprocedural Scalar Reduction of Aggregates
      - Full Redundancy Elimination
      - Copy Propagation
      - Merge PHI nodes
      - Dead Code Elimination
      - Early Interprocedural Scalar Reduction of Aggregates
      - Tail recursion elimination
      - Switch conversion
      - Cleanup exception handling
      - Profiling pass
      - Local pure-const analysis
      - Split functions to improve inlining
    - Release SSA names
    - Rebuild call graph edges
    - Inline analysis
  - Tree profiling pass
    - Split functions based on feedback
  - Increase alignment of global arrays to improve vectorization potential
  - Matrix flattening
  - Transactions analysis
  - Thread local storage emulation
- Regular IPA passes
  - IPA whole program visibility analysis
  - Read profile data and propagate frequencies across the call graph
  - Interprocedural Constant propagation
  - Merge multiple constructors destructors
  - Inlining with whole program knowledge
  - Pure const analysis
  - Static variables usage analysis
- Link-Time Optimization (LTO) generation passes
  - IPA LTO output GIMPLE

    - IPA LTO finish output
- Late IPA passes
    - Alias analysis
- All passes
    - Fixup control flow graph
    - Lower exception handling dispatch instructions
    - All optimizations
        - Remove call graph callee edges
        - Remove predictions no longer used
        - Rename SSA copies
        - Complete unroll inner loops
        - Conditional Constant Propagation
        - Forward propagation of single-use variables
        - Conditional dead call elimination
        - Build alias analysis
        - Return slot optimization
        - PHI node propagation
        - Full Redundancy Elimination
        - Copy propagation
        - Merge PHI nodes
        - Value Range Propagation
        - Dead Code Elimination
        - Transform conditional stores into unconditional ones
        - Combine if-expressions
        - PHI Optimizations
        - Tail recursion
        - Loop Header Copy
        - Optimization of stdarg function
        - Lower complex Arithmetic
        - Scalar Replacement of Aggregates
        - Rename SSA copies
        - Dominator optimizations
        - Eliminate Degenerate PHIs
        - Dead Store Elimination
        - Reassociation
        - Dead Code Elimination
        - Forward propagation of single-use variables
        - PHI Optimizations

- Optimize all builtin objec size computations
- String length optimizations
- Conditional constant propagation
- Copy propagation
- Optimization for sin/cos operations
- Find manual byte swap and transform them to the builtin
- Split critical edges
- Partial Redundancy Elimination
- Code sinking
- Tree loop optimizations
  - Tree loop initialization
  - Loop Invariant Motion
  - Copy propagation
  - Dead Code Elimination in loops
  - Loop Unswitching
  - Constant Propagation using SCEV
  - Record bounds of the loops
  - Check data dependencies
  - Loop Distribution
  - Copy propagation
  - Graphite optimizations
    - Graphite linear transforms
    - Loop Invariant Motion
    - Copy propagation
    - Dead Code Elimination in loop
  - Induction Variable canonicalization
  - If conversion
  - Vectorize
    - Dead Code Elimination in loop
  - Predictive commoning
  - Complete unrolling
  - SLP Vectorization
  - Parallelize loops
  - Array prefetching
  - Induction variables optimizations
  - Loop Invariant Motion
- Lower vector operations to scalar operations
- Common Subexpression Elimination using math reciprocals

    - Reassociation
    - Value Range Propagations
    - Dominator optimizations
    - Eliminate degenerate PHIs
    - Aggressive Dead Code Elimination (take branches into account)
    - Tail duplication for superblock scheduling
    - Dead Store Elimination
    - Forward propagation of single-use variables
    - PHI Optimizations
    - Fold builtin functions
    - Optimize widening multiplications
    - Tail Call Elimination
    - Rename SSA copies
    - Unpropagate edge equivalences
    - Local pure const analysis
  - Transactional memory pass
    - Mark transactional memory
    - Transactional memory optimization
    - Replace TM statements with builtins
  - Lower complex arithmetic in O0
  - Cleanup exception handling
  - Lower RESX statements
  - Return Value Optimization
  - mudflap pass 2
  - Cleanup CFG
  - Warn about non-void function with no return
  - Convert to Register Transfer Language (RTL)
  - Rest of compilation
    - Init function to RTL
    - Cleanup the CFG in RTL
    - Generate real code for Exception Handling
    - Emit initial value sets
    - Unshare RTL structures
    - Instantiate virtual regs
    - Initialize RTL data structures
    - Cleanup CFG
    - Decompose multi-word registers
    - Set up data-flow framework

- Common Subexpression elimination
- Forward propagation of single-use variables
- Global Copy/Constant propagation
- Partial redundancy elimination
- Code Hoisting
- Global Copy/Constant propagation
- Store motion
- Common Subexpression elimination
- If Conversions
- Register Analysis
- Loop analysis
  - Loop Init
  - Loop Invariant Motion
  - Loop unswitching
  - Loop Unroll and Peeling
  - Looping instructions optimizations with low-overhead instructions
- Splits independent use of each pseudo register
- Constant/copy propagation
- Common Subexpression Elimination
- Dead Store Elimination
- Forward Propagation address
- Improve auto-inc and auto-dec instructions
- Initialize all uninitialized to 0 before usage
- Ud-Chain based Dead Code Elimination
- Combine instructions
- If conversion after combine
- Partition blocks in hot/cold sections
- Move register to improve register allocation
- Leave CFG layout mode
- Split instructions
- Decompose multiword registers
- Initialize data-flow framework
- Determine is stack pointer is constant
- Mode Switching optimizations
- Match asm constraints
- Swing Modulo Scheduling
- Integrated Register Allocation
- Reload pass

  - Post reload pass
    - Common Subexpression Elimination
    - Global Common Subexpression Elimination
    - Split all instructions
    - Redundant Extension Elimination
    - Compare elimination
    - Branch target register Load Optimization 1
    - Generate prologue and epilogue for thread
    - Dead Store Elimination
    - Combine stack adjustments
    - Peephole Optimizations
    - If Conversion after reload
    - Register renaming
    - Copy propagation on hard registers
    - Fast Dead Code Elimination
    - Basic Block Reordering
    - Branch target register Load Optimization 2
    - Leaf regs analysis
    - Split instructions
    - Instruction scheduling
    - Stack registers
      - Split instructions
      - Stack registers optimization
    - Compute alignments
    - Duplication blocks containing computed gotos
    - Variable tracking
    - Free the CFG
    - Machine-dependent reorganization
    - Cleanup barriers
    - Fill delay slots
    - Split instructions
    - Convert eh region ranges to individual instructions
    - Shorten branches
    - Set nothrow function flags
    - Output dwarf2 frame
    - Turn the RTL into assembly
  - Finish data-flow framework
- Clean state

Bibliography

[Callahan1991] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS IV, pages 40–52, New York, NY, USA, 1991. ACM.

[Carr1994] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 252–262, New York, NY, USA, 1994. ACM.

[Chen2010] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 42–52, New York, NY, USA, 2010. ACM.

[Cytron1991] 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 Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[Ferdinand1997] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 37–46, 1997.

[Fraboulet2001] Antoine Fraboulet, Karen Kodary, and Anne Mignotte. Loop fusion for memory space optimization. In *Proceedings of the 14th international* *symposium on Systems synthesis*, ISSS '01, pages 95–100, New York, NY, USA, 2001. ACM.

[Ghosh1999] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, July 1999.

[Golovanevsky2007] O. Golovanevsky and A. Zaks. Struct-reorg: current status and future perspectives. In *Proceedings of the GCC Developers' Summit*, pages 47–56, 2007.

[Gove2007] Darryl Gove and Lawrence Spracklen. Evaluating the correspondence between training and reference workloads in spec cpu2006. *SIGARCH Comput. Archit. News*, 35(1):122–129, March 2007.

[Hagog2005] Mostafa Hagog and Caroline Tice. Cache aware data layout reorganization optimization in gcc. In *Proceedings of GCC Summit 2005*, pages 69–92, 2005.

[Hendren92] Laurie J. Hendren, Chris Donawa, Maryam Emami, Guang R. Gao, Justiani, and Bhama Sridharan. Designing the mccat compiler based on a family of structured intermediate representations. In *In Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, number 757 in LNCS*, pages 406–420. Springer-Verlag, 1992.

[Hundt2006] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. Practical structure layout optimization and advice. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 233–244, Washington, DC, USA, 2006. IEEE Computer Society.

[Intel2012] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-044US. August 2012.

[Jeremiassen1995] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 179–188, New York, NY, USA, 1995. ACM.

[Levin2008] Roy Levin, Ilan Newman, and Gadi Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In

Proceedings of the 3rd international conference on High performance embedded architectures and compilers, HiPEAC'08, pages 291–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[Levinthal2008] David Levinthal. Cycle Accounting Analysis on Intel® Core™ 2 Processors. Technical report, Intel Corp., 2008.

[Manjikian1997] Naraig Manjikian. *Program transformations for cache locality enhancement on shared-memory multiprocessors.* PhD thesis, Toronto, Ont., Canada, Canada, 1997.

[Marchal2004] Paul Marchal, José Ignacio Gómez, and Francky Catthoor. Optimizing the memory bandwidth with loop fusion. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pages 188–193, New York, NY, USA, 2004. ACM.

[McKinley1996] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[Papaux2012] Geoffrey Papaux. *Perf-based profiles for GCC.* Bachelor's thesis, Department of Computer Science, College of Engineering and Architecture of Fribourg, 2012.

[Probert1982] R.L. Probert. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering*, 8:34–42, 1982.

[Ramasamy2008] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. Feedback-directed optimizations in gcc with estimated edge profiles from hardware event sampling. In *Proceedings of GCC Summit 2008*, pages 87–102, 2008.

[Rubin2002] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. *SIGPLAN Not.*, 37(1):140–153, January 2002.

[Singhai1996] Sharad Singhai and Kathryn Mckinley. Loop fusion for data locality and parallelism. In *In Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems, New Paltz*, pages 148–150, 1996.

[Sjodin2009] Jan Sjödin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop. Design of graphite and the Polyhedral Compilation Package. In *GCC Developers' Summit*, Montréal, Canada, June 2009.

[Soffa2011] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 888–891, New York, NY, USA, 2011. ACM.

[Srikant2007] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition.* CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2007.

[Tolubaeva2012] M. Tolubaeva, Yonghong Yan, and B. Chapman. Compile-time detection of false sharing via loop cost modeling. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 557 –566, may 2012.

[Weaver2008] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *IISWC*, pages 141–150, 2008.

Index

Glossary

Basic Block A block of instructions that has one entry point(the first instruction) and one exit point(the last instruction). Highly used in compilers optimizations due to its desirable properties.

CFG Control Flow Graph. Graph representation of all paths through basic blocks. A vertex represents a Basic Block and an edge represents a possible path.

denormal A number smaller than the smallest normal floating point number. They are represented with a zero exponent and leading zero bits in the mantissa. .

dominator A basic block D is a dominator of a basic block A if every path from the entry reaching A has to pass through block M. The entry block is a dominator of all the blocks..

heuristic technique designed to solve a problem that ignores whether the solution can be proven to be correct.

L1 Level 1 Cache. It is a very fast cache used to increase memory bandwith.

L2 Level 2 Cache. It is a fast cache used to increase memory bandwith.

LLC Last Level Cache. It is the last cache before the main memory. Typically, it is the L2 or L3 cache.

TLB Transaction Lookaside Buffer. This cache is used to speed up virtual-to-physical memory conversion for addresses.

UD Chain Use-Definition Chain. A use (U) of a variable, and all the definitions (D) of that variable that can reach that use.

Acronyms

AST Abstract Syntax Tree.

CFG Control Flow Graph. *Glossary:* CFG.

FDO Feedback Directed Optimization.

Gooda Generic Optimization Data Analyzer.

IR Intermediate Representation.

LBR Last Branch Record.

LLC Last Level Cache. *Glossary:* LLC.

LTO Link-Time Optimization.

PEBS Precise Event-Based Sampling.

PGO Profile Guided Optimization.

PMU Performance Monitoring Unit.

RTL Register Transfer Language.

SSA Static Single Assignment.

List of Figures

List of Tables