A study on compiler flags and performance events

CERN openlab Technical Report

Mirela-Madalina Botezatu
mirela-madalina.botezatu@cern.ch
Supervisor: Andrzej Nowak

01.09.2012

Table of Contents

1 Introduction

CERN openlab is a partnership between CERN and leading IT companies. CERN gets access to unreleased, cutting edge technology and, on the other side, the partners get access to CERN's expertise and a demanding test environment for their products. We collaborate with Intel in the Platform Competence Centre of CERN openlab. The work of the PCC addresses issues such as power and computing efficiency, benchmarking, optimization, multi-threading and multi-core scalability or high-speed networking.

Compilers are the bridge between software and hardware, and the role they play in satisfying real-time and performance constraints is crucial. The processor architecture controls the ability of the compiler to efficiently generate code that can ultimately bring speed optimization. In order to have a good comprehension on how the software maps onto the running architecture, performance events can be used as excellent indicators.

Accuracy, reproducibility and speed in software are often conflicting objectives. Enabling the appropriate compiler switches can be very helpful in controlling these tradeoffs. The compiler will attempt to optimize the binary and/or the size of the code at the cost of compilation time and possibly the ability to debug the program. But which compiler and further, which compiler flags to choose among the hundreds provided? In the first part of the study we show results from a synthetic set of benchmarks from two major x86 compilers.

Can we quickly identify the performance bottlenecks which exist in the code? Which compiler flags are likely to alleviate which performance issue, and at what cost? We try to answer some of these questions through a set of statistical techniques. We filter out a subset of flags (of the Intel 13.0.1 compiler) that are likely to bring performance gains, but unlike in other studies, we don't compromise the accuracy and reproducibility of the results. We select a set of benchmarks to evaluate code runs with different combinations of flags enabled.

We use performance events to identify the performance bottlenecks present in those benchmarks and eventually we attempt to associate them with the compiler flags that are likely to alleviate these issues.

2 State of the art

In [1], Cavazos et al. used machine learning to build a model that can automatically choose adequate compiler optimization flags for a program. For building the model, the authors used logistic regression[1] which is not a computationally expensive technique. We found it very interesting to explore how this idea maps onto modern technology and software capabilities. We used the latest compiler release from Intel (ICC 13.0.0), classic performance events and also events that were not supported before. However we involve a different machine learning technique. Instead of building a logistic regression model per flag we use random forests[2], so that the combined effect of different enabled flags is also considered.

ACOVEA[3] is a tool developed by Scott Robert Ladd which implements genetic algorithms to select the best compiler flags to build singular algorithms using GCC. In [2] several explanatory techniques are applied on performance events to answer questions such as which performance events have similar information or whether they provide the same information across different parallel tasks or which of them help differentiate between tasks. In our study we also involve explanatory techniques like the scatter matrix of performance events, or Principal Component Analysis.

In [3] S. Bird et.al illustrate the achievements in terms of performance brought by the new features like macro-fusion and micro-fusion in Intel's Woodcrest processor. Looking at branch mispredictions per KI,[4] L1D cache misses per KI and L2 misses per KI they see that L2 misses have the highest impact on performance (0.96 correlation coefficient). They compare Woodcrest with some predecessors with similar architectural features. They correlated the increase in performance with the percentage of fused operations and they noticed a high correlation between the increase in performance of Woodcrest over Yonah or a NetBurst architecture based processor (Intel Pentium Extreme Edition 965) and the number of macro-fusions for integer benchmarks.

Data mining has also been involved in estimating the power consumption by looking at performance events. Stockman et al. [4] used neural networks while Contreras et al. [5] used a linear model for this purpose.

In [6] Wucherl Yoo et al. use the notion of performance pathologies for performance bottlenecks that appear during the execution of a program. They implemented decision trees for performance pathologies identification. Another interesting data mining study applied on performance events is [7], where it is presented how performance counters can be used for fault localization by monitoring the number of instructions retired at function level.

[1] Logistic regression is a type of regression analysis that is used to predict the outcome of a dichotomous dependent variable

[2] Random forests is an ensemble classifier that consists of many decision trees and predicts the class of a categorical variable

[3] Analysis of Compiler Options via Evolutionary Algorithms

[4] KI = A thousand of instructions

3 Benchmarks

In order to perform our analysis we chose a set of 37 benchmarks, some of which stress the CPU and others the I/O subsystem. There are 4 sets, and in addition we also benchmarked a Fast Fourier Transform implementation[5].

1. **High Energy Physics (HEP) benchmarks** – a set of benchmarks developed in openlab, which consists of representative snippets for evaluating the code from CLHEP[6], GEANT4[7], ROOT[8] and STL[9].
2. **Root** benchmarks[10] - official benchmarks for stressing the functionality of ROOT.
3. **Gooda**[11] I/O intensive benchmarks.
4. **Adobe C++ Benchmarks** - a set of C++ benchmarks typically used to quantify how well top compiler vendors implement various C++ operations and language features.

When analyzing the influence of compiler flags one typically looks at the runtime performance of the code. It may be interesting, but not critical, to look at compile time as well (as usually the code is compiled once, run many times). To analyze the former - the runtime performance - we use the benchmarks mentioned above and for the latter – compilation time - we use **HEPSPEC06**.

In addition to a well established set of benchmarks, we used the Adobe C++ Benchmarks as they tackle common performance issues encountered in C++ code and represent good optimization challenges for the compiler. In the following table, we list the names of those benchmarks and a brief description for each[12].

Benchmark	**Description**
functionobjects	This test is a demonstration of the performance of function pointers, functors, and native comparison operators. Some compilers have difficulty instantiating simple functors.
simple_types_loop_invariant	A test to check if the compiler will move loop invariant calculations out of the loop. Most compilers have room for improvement.
stepanov_vector	Usage of pointers to vector iterators and usage of reverse iterators. This tests the compiler supplied STL implementation in addition to the compiler itself.

[5]The FFT implementation is both I/O intensive and CPU intensive benchmark and it tests the processor's performance in converting domain data into frequency domain data. We chose the implementation of Don Cross - http://groovit.disjunkt.com/analog/time-domain/fft.html

[6] http://proj-clhep.web.cern.ch/proj-clhep/

[7] http://geant4.cern.ch/

[8] http://root.cern.ch/drupal/

[9] http://en.wikipedia.org/wiki/Standard_Template_Library

[10] A list of benchmarks from the ROOT standard distribution kit: http://root.cern.ch/drupal/content/benchmarking

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

[12] http://www.nersc.gov/users/computational-systems/hopper/performance-and-optimization/compiler-comparisons/

simple_types_constant_folding	A test to check if the compiler will correctly fold constants and simple constant math for simple types.
loop_unroll	Test to check if compilers will correctly unroll loops to hide instruction latency. Some compilers have problems expanding the templates, and most compilers have problems correctly unrolling the loops for best performance.
stepanov_abstraction	A value wrapped in a structure or class should not perform worse than a raw value. Through this test we measure the performance penalty caused by the use of data abstraction in C++ programs.

Note:
The machine used for studies is the following:

Westmere [Intel(R) Xeon(R) CPU X5650 2713 MHz, 24 cores , 2 sockets, Hyper-Threading on, Cache size: 12288KB, RAM size: 47 GB]

4 A comparison of compiled binary speed of ICC and GCC

As we want to use tools that help us most in producing fast, optimal code, we compared the performance of the two most popular x86 compilers– the Intel compiler and the GNU compiler on Adobe benchmarks. As the figure of merit we used the execution time measured in seconds. We used only the Adobe benchmarks because the code is written to address directly relevant performance issues (both in the sense that they are often encountered and that they represent real challenges for compilers in their attempt at optimizing the code).

The compiler versions used for the analysis are:

- ICC 13.0.1
- GCC 4.6.3

We ran the benchmarks compiled with the two compilers for two optimization levels: O2 and O3.

Note:

The optimization levels O2 and O3 are similar between ICC and GCC, but different in important ways on the two compilers (for example, ICC allows unsafe floating-point optimizations at –O2 (and –O3) and GCC doesn't even at –O3). They are each a "combination" of various internal individual options and the driver passes those individual options to the compiler. Some examples of differences between the O2 and O3 optimization levels for the two compilers (for the versions mentioned above):

- ICC enables inlining at O2 whereas GCC enables it at O3.

- ICC at O2 optimization level has inlining and other interprocedural optimizations within a source file, vectorization. Vectorization and most inlining is enabled in GCC only at the O3 optimization level.
- GCC enables "-fstrict-aliasing" (enforces strict aliasing rules) starting from O2 whereas ICC doesn't enable it even at O3.
- Loop unrolling is enabled starting from O2 with ICC whereas in GCC at O2 there is the flag “frerun-loop-opt”, which also enables some loop optimizations, but no loop unrolling.
- ICC has optimized math library functions by default.

The results from the runs are presented in the following table:

Benchmark	Exec. time **GCC –O2**	Exec. time **ICC –O2**	**ICC Gain**	Exec. time **GCC –O3**	Exec. time **ICC –O3**	**ICC Gain**
functionobjects.cpp	245.05	238.60	**2%**	240.97	240.58	**0%**
loop_unroll.cpp	383.04	198.63	**48%**	388.93	167.63	**56%**
Simple_types_constant_folding.cpp	104.33	155.6	**-49%**	97.05	155.79	**-59%**
Simple_types_loop_invariant.cpp	354.92	245.38	**30%**	333.19	245.13	**26%**
Stepanov_abstraction.cpp	248.99	213.49	**14%**	245.77	234.73	**4%**
Stepanov_vector.cpp	301.38	214.303	**28%**	303.06	228.004	**24%**

Adobe benchmarks - execution time measured in seconds

We compare the execution time obtained for these runs and we observe that ICC outperforms GCC in 5 out of the 6 benchmarks, the speedup obtained by compiling with ICC ranging from 1% to 56%. However, we see that GCC appears to deal better with folding constant mathem atical expressions.

5 Compiler flag roles and restrictions

As mentioned before, the selection of the proper set of compiler optimization flags is subject to a judicious choice. It is very costly in terms of time to analyze all the flags in all possible combinations. In order to ease this process, we selected optimization flags that target different optimization paths, that can strongly impact performance, and whose effects can be identified through the performance events.

Note: +EXPAND

- We did not include those flags that disregard strict standards compliance[13].
- We did not include flags that are enabled by default.
- We did not include “tune for this architecture” switches

Based on the documentation provided by Intel [9] and the advice received from Intel experts, we selected the following options:

[13]However, we did not use “–fp-model strict” for ICC to get standard compliant floating-point behavior, and by default “fp-model fast=1” is enabled, which enables more aggressive optimizations on floating=point calculations

Flag	Description
-O3	O2 optimizations plus more aggressive optimizations for maximum speed like: • Loop unrolling and instruction scheduling • Code replication to eliminate branches • Padding the size of power two arrays to allow more efficient cache use
-fno-inline-functions	It is the opposite of finline-functions which is enabled in O2 and O3
-inline-forceinline	Specifies that an inline routine should be inlined whenever the compiler can do so. Because C++ member functions whose definitions are included in the class declaration are considered inlinefunctions by default, using this option will also make these member functions "forceinline" functions. The compiler will not inline if that creates problems: for example, a recursive function is inlined into itself only once.
-nolib-inline	Disables inline expansion of standard library or intrinsic functions. (It prevents the unexpected results that can arise from inline expansion of these functions, like floating-point computations inconsistency.)
-unroll-aggressive	This option enables aggressive, complete unrolling for loops with small constant trip counts.
-funroll-all-loops	Unroll all loops even if the number of iterations is uncertain when the loop is entered.
-falign-functions	A align functions on an optimal byte boundary.
-ansi-alias	Assume that the program adheres to ISO C Standard aliasing rules. This allows the compiler to optimize more aggressively. If the code does not adhere to these rules then it can cause the compiler to generate incorrect code. Note: Our benchmarks conform to these rules
-opt-streaming-stores always	Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound.
-opt-class-analysis	Determines whether C++ class hierarchy information is used to analyze and resolve C++ virtual function calls at compile time. The option is turned on by default with –ip or –ipo compiler options, enabling improved C++ optimization.
-opt-ra-region-strategy=routine	The register allocator creates a single region for each routine.
-opt-ra-region-strategy=block	The register allocator partitions each routine into one region per basic block.
-ip	Enables additional interprocedural optimizations for single-file compilations.
-ipo	Enables interprocedural optimizations between files. When this flag is enabled, the compiler performs inline function expansion for calls to functions defined in separate files.
-opt-prefetch=4	Enables prefetch insertion optimization. We test with opt-prefetch=4 so that it performs more aggressive prefetching.

-opt-block-factor=2	Loop-blocking factor=2. Loop blocking optimization is part of the High Level Optimizations in Intel compiler. It is available when the optimization level is higher or equal with –O3.
-opt-block-factor=16	Loop blocking factor = 16.

We decided to exclude the compiler optimization flags that are slightly risky[14], like no-prec-div, no-prec-sqrt, -fast-transcedentals. They might introduce inaccuracy and HEP code is sensitive to this issue.

To illustrate the performance impact of unsafe optimizations, we run the "simple types constant folding" benchmark (from Adobe C++ Benchmarks), we extract two of the tested cases ("float constant divide" and "float multiple constant divides") and compare the execution time for the code compiled with –O2 with the execution time for divisions for the code compiled with –fast.

Operation	**Execution Time (seconds) Code compiled with –O2**	**Execution Time (seconds) Code compiled with –fast**	**Speedup**
Float constant divide	62.51	39.55	**1.58**
Float multiple constant divides	184.6	39.42	**4.68**

Code compiled with -O2 vs code compiled with -fast

We see that divide operations can be executed 4.5 times faster when "-fast" is switched on. This flag maximizes the speed across the entire program. It enables a number of "unsafe" optimizations like -ipo, -O3, -no-prec-div, -static.

Another optimization flag that has potential of being helpful but was not considered, because it is risky, is "-fno-alias" and its equivalent for functions only "- fno-fnalias" (for functions), which assumes less strict rules. Aliasing implies writing to a location in memory with more than one pointer to that address. By strict aliasing rules we denote rules that specify that memory references are not allowed to the same memory locations via separate pointers. If the piece of code does not conform to the alias rule in effect, the compiler might generate optimizations that modify the intended semantics of the program. This in turn can lead to incorrect results or runtime failures. We leave the decision of enabling strict aliasing rules to be made by the programmer.

One has to keep in mind the fact that possible aliasing can sometimes prevent the compiler from pipelining the instructions, or from benefiting from the parallelism capabilities in the processor.

An alternative for dealing with aliasing is "opt-multi-version-aggressive" which forces the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement. It is well suited for situations where one knows one can't use -ansi-alias but not every part of the code violates the alias rules. This option is "assumption-free", as different versions of the loop may be generated based on run time dependence testing, alignment and checking for short/long trip counts. When enabled, this option

[14] Note that "-fp-model fast=1" which is enabled by default with ICC, may also alter the accuracy of floating-point computations

will trigger more versioning at the expense of creating more overhead to check for pointer aliasing and scalar replacement.

An eye should be kept on feedback-driven optimization, enabled by the "-pgo" flag which was not included in the study since it does not fit our methodology. It is a powerful technique that tries to optimize the most heavily executed paths in the program. It is deterministic, as it does not use actual execution time to tune the optimization. It is input dependent - the input for the run, based on which the default heuristics are tuned for various optimizations, must be representative.

5.1. A study of performance benefits with particular combinations of compiler flags

Since some flags interact with others, we analyze both the effects of a flag when used individually and in combination with other compatible flags

We split the benchmarks in two subsets: CPU intensive and I/O intensive. 27 benchmarks fall into the former category and 10 into the latter.

We run the benchmarks with one flag switched on and the rest off, then with two flags switched on and the rest off (all the pairs), and then with three flags switched on and the rest off. These combinations are subject to some predefined constraints, e.g. we don't combine flags that are enabled automatically by other flags, we don't combine flags when one can overwrite the effect of the other. As a result we have 786 possible configurations to analyze.

In the following we will present performance results obtained after running the benchmarks with the different combinations. We assume a performance gain if the code runs faster than the code compiled with –O2. We analyze all the runs where the execution time was *at least 1%* faster than the execution time measured for the code compiler with "–O2". For our 37 benchmarks run with the previously mentioned combinations of flags, we have 4421 cases of at least 1% increase in performance out of 29082 cases. We check which flags appeared most often enabled for the cases where we have an increase in performance of at least 1%. We also check which flags appeared most often as bringing performance degradation. In the following table we list "counts" per flag here "counts" represent the number of times there was an increase in performance and the flag was enabled.

Compiler flag	Counts	Compiler flag	Counts
O3	963	Opt-streaming-stores-always	694
Ipo	951	Ansi-alias	686
Opt-ra-region-strategy=routine	821	Opt-prefetch=4	674
Ip	761	Faling-functions	657
Opt-ra-region-strategy=block	760	Unroll-aggressive	652
Funroll-all-loops	753	fno-inline-functions	628
Nolib-inline	740	Opt-block-factor=16	616
Inline-forceinline	738	Opt-block-factor=2	608
Opt-class-analysis	700		

Flags bringing a performance gain sorted by counts

5.1.1 Observations on the CPU intensive subset of benchmarks

For the CPU intensive subset of benchmarks, in 17 out of 27 benchmarks we have an increase in performance of at least 1% for at least one combination of compiler flags (including the “one flag only” case).

Combinations of 1 compiler flag

The flags enumerated below are those that appear in the largest number of benchmarks with a performance gain with respect to the execution time of the code compiled with “–O2”.

Flag	Frequency
Opt-ra-region-strategy=routine	8/17
O3	7/17
Opt-class-analysis	7/17

Combinations of 2 compiler flags

The pairs enumerated below are those combinations of two flags that appear in the largest number of benchmarks with a performance gain with respect to the execution time of the code compiled with “–O2”.

Combination of two flags		Frequency
Opt-ra-region-strategy=routine	ipo	10/17
Opt-ra-region-strategy=routine	ip	9/17
Opt-ra-region-strategy=routine	Opt-block-factor=2	8/17
ipo	Opt-class-analysis	8/17
O3	Opt-ra-region-strategy=routine	8/17
Opt-ra-region-strategy=routine	Inline-forceinline	8/17

Combinations of 3 compiler flags

The triples enumerated bellow are those combinations of three flags that appear in the largest number of benchmarks with a performance gain.

Combination of three flags			Frequency
Opt-ra-region-strategy=routine	Opt-block-factor=2	Ipo	10/17
Opt-ra-region-strategy=routine	Opt-prefetch=4	ipo	9/17
Opt-ra-region-strategy=routine	ip	Ipo	9/17
Opt-ra-region-strategy=routine	Inline-forceinline	Ipo	9/17
Opt-ra-region-strategy=routine	O3	Ip	9/17

Top speed in the CPU intensive benchmark subset:

For each benchmark we select the flags combination with best speedup. Then we see which flags appear most often across all benchmarks. (the speedups range from 1.008 to 1.87) . We notice it is more probable to attain top speed if we enable ip, forced inlining, and we choose suitable register allocation. One should also consider the benefits of unrolling, blocking and prefetching.

Flag	Frequency
Inline-forceinline	9
Opt-ra-region-strategy=block	7
Ip	7
Ipo	7
Opt-prefetch=4	7
Opt-block-factor=2	6
Falign-functions	6
Unroll-aggressive	6
Opt-ra-region-strategy=routine	5
Ansi-alias	5
Opt-class-analysis	4
O3	3

Flags bringing top speed in the CPU intensive benchmark subset, sorted by frequency (the nr of times it appeared across the top combinations of flags - those with best speedup per benchmark-)

5.1.2 Observations on the I/O intensive subset of benchmarks

For the CPU intensive subset of benchmarks in 5 out of 10 benchmarks we have an increase in performance of at least 1% for at least one combination of the compiler flags (including the "one flag only" case).

Combinations of 1 compiler flag:

The flags enumerated bellow are those that appear in the largest number of benchmarks with a performance gain with respect to the execution time of the code compiled with "–O2".

Flag	Frequency
O3	4/5
Opt-prefetch=4	3/5
Ansi-alias	3/5
Ip	3/5

Combinations of 2 compiler flags:

The pairs enumerated below are those combinations of 2 flags that appear in the largest number of benchmarks with a performance gain with respect to the execution time of the code compiled with "-O2".

Combination of two flags		Frequency
Opt-prefetch=4	Ipo	4/5
Ip	Unroll-aggressive	3/5
O3	opt-block-factor=16	3/5
O3	Opt-class-analysis	3/5
O3	Ansi-alias	3/5

Combinations of 3 compiler flags:

The triples enumerated bellow are those combinations of three flags that appear in the largest number of benchmarks with a performance gain. The performance gain is considered relatively to the execution time of the code compiled with "–O2".

Combination of three flags			Frequency
Ansi-alias	Ipo	Opt-streaming-stores always	4/5
O3	Opt-prefetch=4	Ipo	4/5
Fno-inline-functions	Ipo	Ansi-alias	3/5
ansi-alias	Ip	funroll-all-loops	3/5

Top speed in the I/O intensive subset:

For each benchmark we select the flags combination with best speedup. Then we see which flags appear most often across all benchmarks. (speedups range from 1.0001 to 1.31). We notice one is likely to attain top speed in I/O benchmarks if aggressive prefetching is enabled.

Flag	Frequency
Opt-prefetch=4	5
Unroll-aggressive	3
Funroll-all-loops	3
Opt-ra-region-strategy=block	2
Ip	2
Ipo	2
O3	2
Opt-block-factor=2	2
Opt-block-factor=16	2

Flags bringing top speed in the I/O intensive benchmark subset, sorted by frequency (the nr of times it appeared across the top combinations of flags - those with best speedup per benchmark-)

Observations:

The body of a loop tends to be executed frequently. It is very important to apply beneficial transformations and to apply appropriate register allocation technique, as this typically greatly influences the number of memory accesses. From the data mentioned above we can see that register allocation interferes heavily with loop unrolling and with the blocking factor.

Register allocation is also affected by software prefetching. For our benchmarks this interaction brought a performance again. However, this might also increase the register pressure and lead to a decrease in performance, as stated by Shrewsbury and Norris in [10]. The authors identified two main reasons for a potentially unfavorable interaction. First, there are additional instructions inserted in order to calculate the address to fetch. These instructions will use some temporaries that should be stored in registers and this increases the competition for registers. The second reason is represented by the loop transformations needed for scheduling the prefetched instructions. They are likely to increase code size and complicate the interference graph based on which registers are assigned to temporary variables.

Performance decrease

We see that the option “opt-streaming-stores=always” and “nolib-inline” do not seem to have a positive performance impact on our benchmarks…

We analyze all the runs where performance degradation is noticed. We label a configuration as bringing a performance decrease, if the execution time for the code compiled using that configuration was at least 1% longer than the execution time measured for the code compiled with “–O2”. For our 37 benchmarks ran with the previously mentioned combinations of flags, we have 4515 cases of a decrease in performance out of 29082 cases, approximately 15% of the total.

If we check which flags appeared most often enabled for these cases we have the following results:

Flag	**Frequency**	**Flag**	**Frequency**
Opt-streaming-stores-always	1071	Ansi-alias	686
Nolib-inline	1004	Opt-prefetch=4	675
O3	838	Funroll-all-loops	673
Ipo	822	Inline-forceinline	665
Opt-ra-region-strategy=block	818	Unroll-aggressive	656
fno-inline-functions	773	Opt-class-analysis	647
Opt-ra-region-strategy=routine	757	Opt-block-factor=16	590
Ip	710	Opt-block-factor=2	586
Falign-functions	688		

Compiler flags that caused performance degradation

From the table above we can see that “nolib-inline”, “O3” and “opt-streaming-stores-always” appear quite frequently in the combinations of flags that have a negative impact on the performance.

The flags that, alone, lead to approx 15% decrease in performance:

- opt-streaming-stores always (*“stepanov_vector”*)
- nolib-inline (*“testSTLlist”, “testSTLvect”*)

The combinations of 2 that lead to at least 20% decrease in performance in our benchmarks:

- unroll-aggressive + opt-streaming-stores always (*“testTGeoBBox”,”testTRandom”*)
- nolib-inline + opt-block-factor=2 (*“testTGeoBBox”, “testSTLvect”*)
- nolib-inline + opt-ra-region-strategy=block (*“testSTLlist”, “testSTLvect”*)
- opt-streaming-stores-always + nolib-inline (*“testSTLlist”, “testSTLvect”*)

The combinations of 3 that lead to at least 30% decrease in performance in our benchmarks:

- nolib-inline + opt-streaming-stores always + opt-prefetch=4 (*“testTrandom”, “testSTLvect”*)
- nolib-inline + opt-streaming-stores always + ip (*“testTrandom”, “testSTLvect”*)
- inline-forceinline+nolib-inline+unroll-aggressive (*“testTrandom”, “testSTLvect”*)
- fno-inline-functions + nolib-inline + ansi-alias (*“testSTLlist”, “testSTLvect”*)

6 A study of the influence of compilation flags on the compilation time

Real world applications can consist of millions of lines of code in hundreds of source files, which can take a considerable amount of time to build. Gains in execution time achieved by enabling different compiler flags are often at the expense of compilation time. That's why it is important to analyze the compile-time cost of the flags which improve performance. We will do this by measuring the compile time with each flag from our pool on a representative benchmark – HEPSPEC06.

Using the Intel compiler involves going through three main stages:

1. The compiler is loaded
2. The compiler tries to find a license
3. The compiler compiles the code

The time spent by the compiler trying to find a license depends on various factors like whether one is using a local license or a license server or whether there were a lot of "old" licenses around. It is important how many licenses it has to look through before finding a valid one. Having to "check out" a license from a multi-unit license takes time.

In order to diminish the importance of the time spent in phases 1 and 2, we require a workload that spends much more time in phase 3 than in the other ones. This way, we can compare more accurately the time spent compiling with one flag versus another. Therefore we opted for HEPSPEC06, a complex set of test applications intended for measuring CPU performance. HEPSPEC06 is based on the "all_cpp" benchmark subset of the widely used, industry standard SPEC CPU2006 benchmark suite.

We noted the compile time[15] for HEPSPEC06 suite using each flag from the collection, and the results are displayed in the following graph:

[15] Time – total time not just the time spent in step 3.

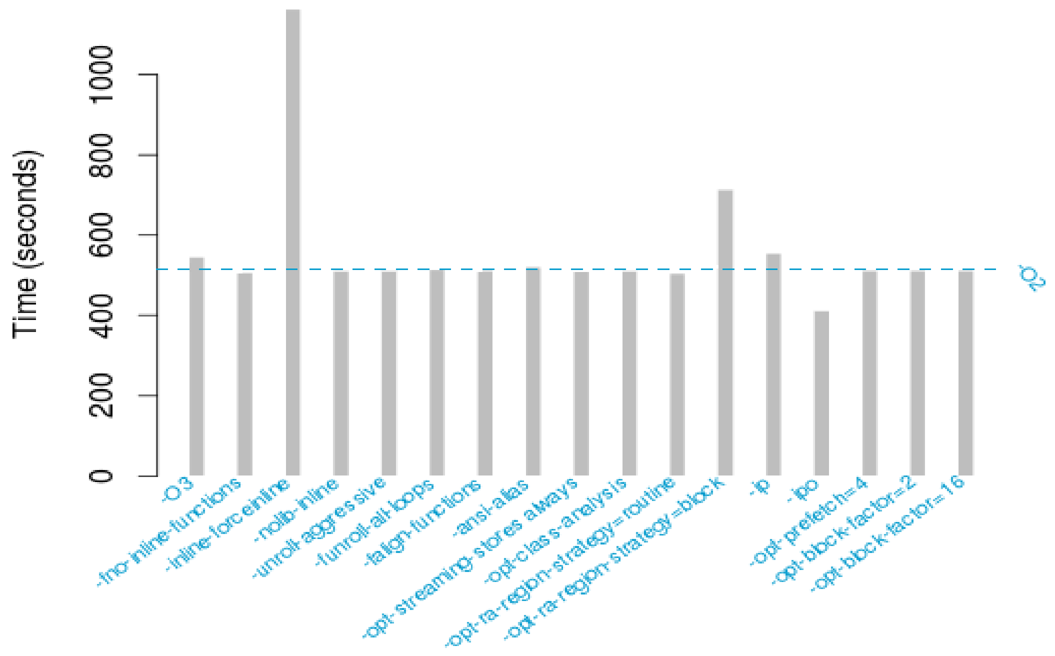

Figure 1: Compilation time relative to -O2 on HEPSPEC06

From the figure above we can see that the “inline-forceinline” flag doubles the time spent in building the code for these tests. The intermediate representation of the code of a function is copied in each place where it is called.

Observations:

C++ inlining is performed at compile time. This implies that if one modifies the code of the inlined function one must recompile all the code using it to ensure that all the changes are propagated where needed. For this flag, “inline-forceinline”, particular attention is needed for the question if the benefits in execution pay off the cost in compile time.

”Opt-ra-region-strategy=block” also brings a 50% increase in compilation time. Register allocation is a costly operation and the time spent in register allocation increases as the number of regions grows. Accordingly, opt-ra-region-strategy=block is the most costly in compile time, whereas at the opposite end we have opt-ra-region-strategy=routine.

Also, we can see that “-ip” brings a little increase in the build time while in the presence of “-ipo” we can observe a decrease in the compilation time.

7 Using machine learning for performance events analysis

The Performance Monitoring Unit consists of a set of registers that are used in counting micro-architectural events from various hardware sources (CPU, memory buffers, pipeline, caches, bus, etc). Collecting and analysing the events is very helpful for investigating the two main causes of poor performance: suboptimal code generation and sub optimal interaction of the code and the micro-architecture.

Performance events are hardware specific data that are used to analyze the interaction of code with microarchitecture specific components. Therefore, the capabilities of the tools used for collecting performance data depend on the features of the CPU. Not all processors come with the same set of performance events, they are specified by the processor manufacturer. Performance events can be classified into the following categories:

- General processor characterization
  - General metrics
  - Microarchitectural efficiency and resource utilization
- On-core memory access
- Off-core memory access

For this study we use the same benchmarks we used in the previous analysis.

In order to collect the counts for the performance events, we used perf[16] which is a tool that brings to the user the various abstractions related to hardware specific capabilities. One can collect counters to monitor the entire system or just on a per process or per thread basis, either in counting mode or by sampling events. We use perf in counting mode, (aka ”perf stat”) that collects event counts during process execution. We also use libpfm[17], a library that provides a mapping between performance event names and their encodings and also is aware of the constraints between them.

The performance events selected for the analysis are the following:

PERFORMANCE EVENT	DESCRIPTION
UNHALTED_CORE_CYCLES	Clock cycles when not halted
INSTRUCTION_RETIRED	Number of instructions retired
UOPS_RETIRED:ANY	Number of micro-ops retired
UOPS_ISSUED:ANY	Number of micro-ops issued
ARITH:CYCLES_DIV_BUSY	Cycles that either the divide or sqrt execu-tion unit was occupied
ARITH:DIV	Divide operations executed
RESOURCE_STALLS:ANY	Resource related stall cycles
BR_INST_EXEC:ANY	Number of branch instructions executed
BR_MISP_RETIRED:ALL_BRANCHES	Number of mispredicted branches retired
BACLEAR:CLEAR	Number of times the front end is re-steered, mainly when the Branch Prediction Unit cannot provide a correct prediction.
L2_RQSTS :IFETCH:HIT	Code requests that hit the L2
L2_RQSTS_IFETCH:MISS	Code requests that miss the L2
ITLB_MISSES	ITLB misses

[16] https://perf.wiki.kernel.org/index.php/Main_Page
[17] http://www.hpl.hp.com/research/linux/perfmon/libpfm.php

DTLB_LOAD_MISSES	DTLB misses
MEM_LOAD_RETIRED:L1D HIT	Retired loads that hit the L1 data cache
MEM_LOAD_RETIRED:L2 HIT	Retired loads that hit the L2 cache
MEM_LOAD_RETIRED:LLC_UNSHARED_HIT	Retired loads that hit valid versions in the LLC
MEM_LOAD_RETIRED:OTHER_CORE L2_HIT_HITM	Memory instructions retired LL3 Cache hit and HITM in sibling core
MEM_UNCORE_RETIRED:LOCAL_HITM	Load instructions retired that HIT modified data in sibling core
MEM_UNCORE_RETIRED:LOCAL _DRAM_AND_REMOTE_CACHE_HIT	Load instructions retired local dram and remote cache HIT data sources
MEM_UNCORE_RETIRED:REMOTE_DRAM	Load instructions retired remote DRAM and remote home-remote cache HITM
MEM_UNCORE_RETIRED:REMOTE_HITM	Retired loads that hit remote socket in modified state
MEM_UNCORE RETIRED:OTHER_LLC_MISS	Retired loads that missed the LLC of other cores

Note: Performance events change from processor generation to generation and they are not publicly validated (there is a degree of uncertainty with respect to what some of them measure).

In the following sections we will explore the data through a set of descriptive statistics techniques.

7.1 A study of performance events correlations

The scatter-plot is an exploratory data visualization technique. More precisely it is a graph where two sets of data are plotted against each other to see if a connection or correlation can be established between them. The scatter-plot matrix is used when we deal with more sets of data – more predictor variables, and we want to see all the pair-wise relations at once, by displaying all the $\binom{n}{2}$ scatter plots in a matrix.

The scatter matrix can give us insight with respect to the following issues:

1. The correlation between performance events and if there is a correlation, its direction (whether they are positively or negatively correlated).
2. Detection of benchmarks with unusual (either too large or either too small) ratios for some performance events.

Examining the correlations between performance events can also be useful for architectural analysis and comparison. In our study we will not discuss this aspect, since we collected counts for the performance events from only a single type of machine. On Figure 2, we show the scatter-plot matrix of performance events collected from running the benchmarks compiled with "-O2" optimization level. On the diagonal we have the names of the performance events, the subdiagonal boxes have the scatter plots together with the best linear fit lines, and on the superdiagonal boxes we have Pearson's correlation coefficient[18].

[18] Pearson's correlation coefficient is a measure of the correlation (linear dependence) between two variables, and it can take values between -1 and 1 inclusive.

We show only the events with a very small coefficient of variation (Table 2). We wanted to see if we have run to run counting variations for the values collected for our performance events. As they will be the explanatory variables for our analysis it is essential to know they can be relied upon. We noticed that two performance events present large run to run variations, and consequently we decided to remove them from the analysis. The approach for seeing run to run variations of events is the following: We ran the entire collection of benchmarks three times, we divided the values from one run by the values obtained from the other run and then we computed the coefficient of variation (standard deviation divided by the mean). Those performance events with their coefficient of variation larger than 0.1 were not included in the creation of the scatter-plot matrix[19]

Event	**Coefficient of variation**
UNHALTED_CORE_CYCLES	0.021
INSTRUCTION_RETIRED	0.004
UOPS_RETIRED:ANY	0.004
UOPS_ISSUED:ANY	0.005
ARITH:CYCLES_DIV_BUSY	0.09
ARITH:DIV	0.08
BR_INST_EXEC:ANY	0.006
BR_MISP_RETIRED:ALL_BRANCHES	0.01
BACLEAR:CLEAR	0.09
L2_RQSTS :IFETCH:HIT	0.081
ITLB_MISSES	0.09
MEM_LOAD_RETIRED:L1D_HIT	0.006
MEM_LOAD_RETIRED:LLC_UNSHARED_HIT	0.04

Performance events with a small coefficient of variation

The correlation between performance events:

Figure 2 demonstrates that the more micro-ops issued are counted for a benchmark the more micro-ops and instructions retired which is rather expected. The number of cycles spent increases with memory events (positive correlation), which again is expected. More importantly, the greater the number of mispredicted branches, the more ITLB misses and hence the more cycles spent by that code to execute (this can be explained by branching causing ITLB misses). It is very improbable that one will miss in the ITLB if one just moves forward from one instruction to another – as modern processors do a lot of instruction prefetching. But when one branches one goes to a new address that is out of the sequence and if this instruction has not been executed recently, then most likely it will not be found in the ITLB, hence it has to be retranslated.

We notice very high correlations between UOPS_RETIRED.ANY and MEM LOAD RETIRED:L1D HIT (0.91) and between ITLB_MISSES and BACLEAR.CLEAR (0.9). A high value for BACLEAR.CLEAR usually indicates that the code has many branches. In turn, ITLB misses can result from correct and incorrect branch prediction.

[19] Those events were mostly costly memory events

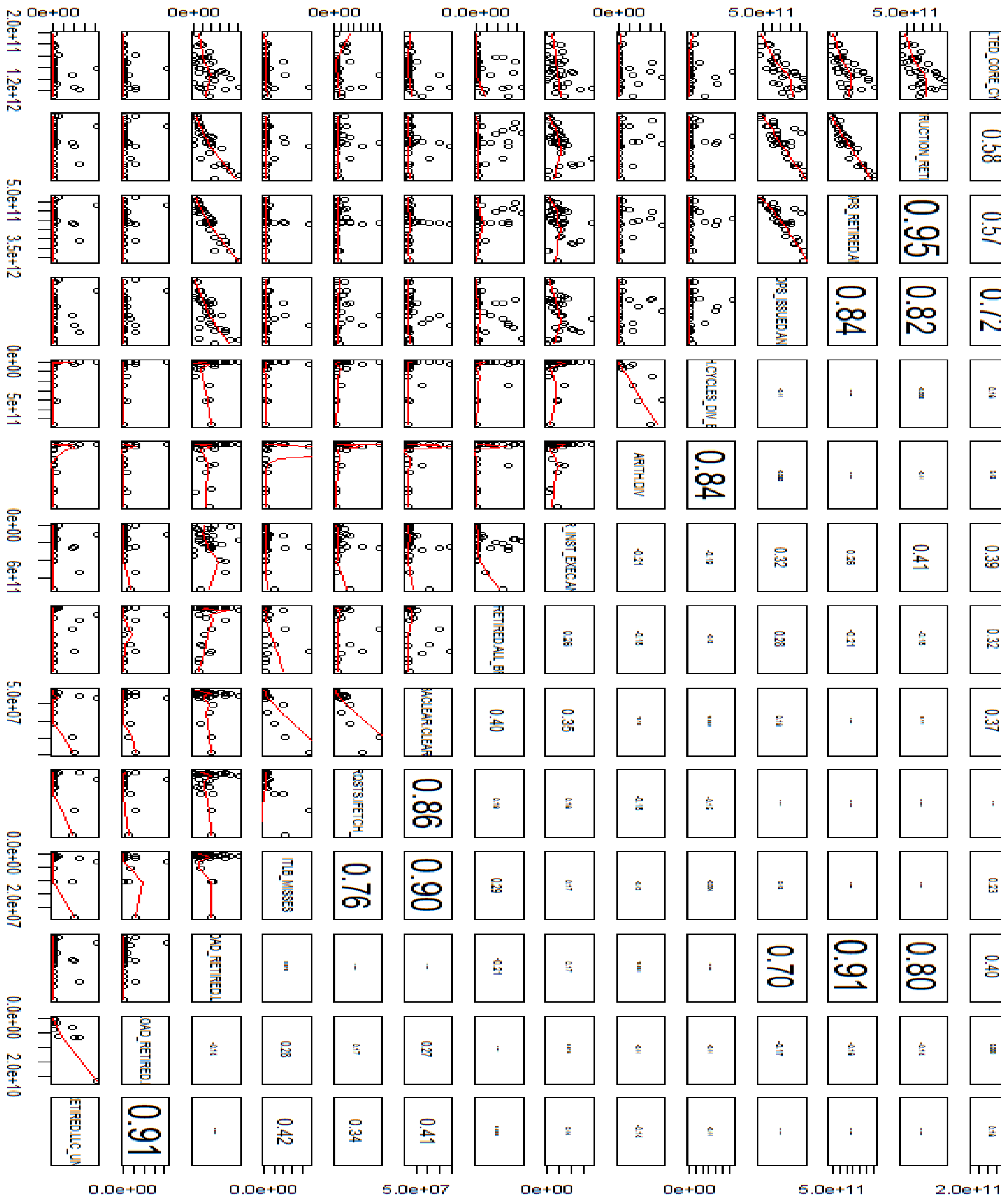

Figure 2 - Scatter-plot matrix

Detection of benchmarks with excessive ratios for some performance events.

If we look closer at the scatter plots we see there are benchmarks that have much larger or much smaller values for the ratios of some performance events. These benchmarks are worth further investigation. For example we discover that for "stressFit" the ratio BACLEAR.CLEAR / BR_MISP_RETIRED_ALL_BRANCHES or BACLEAR.CLEAR / INSTRUCTIONS.RETIRED is two orders of magnitude higher than the average. This creates a case for a profile based analysis.

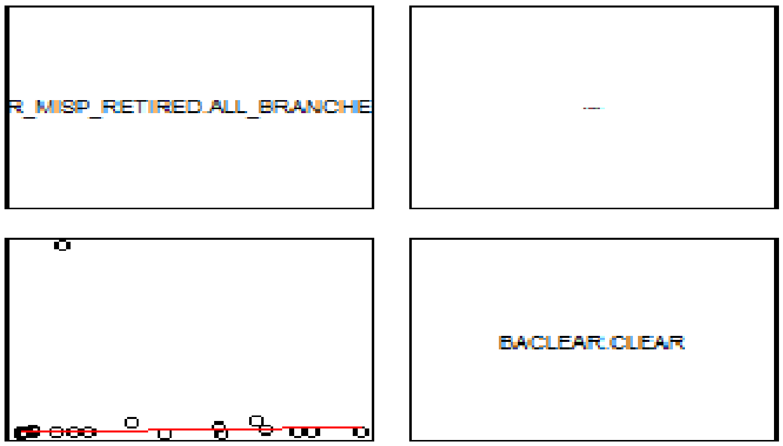

Figure 3 - Scatter-plot matrix zoomed for two events

Similarly, we observe "TestRanluxEngine" (random number generator, many conditional branches) has a very small value for the ratio UOPS_RETIRED / UOPS_ISSUED, approx 0.4 compared to the average 0.9 – 1.1. "TestTGeoArb8" has a very small value for the ratio ARITH_CYCLES_DIV_BUSY / ARITH_DIV, so for the same number of divide operations, fewer cycles are spent by the divide execution unit.

7.2 PCA and varimax rotation

Principal Component Analysis is a technique for dimension reduction, data visualization and compression, latent concept discovery, and preprocessing data in general. We use it in conjunction with varimax rotation which maximizes the sum of the variances of the squared ***factor loadings***. Factor loadings are the correlation coefficients between our data and the factors, and the ***factors*** in turn are the unobserved variables (the eigenvectors corresponding to the largest eigenvalues) resulted from the Singular Value Decomposition of our matrix. We reduce the dimensionality of our data, we extract these principal factors (principal components) and we map the data to this lower dimensional space. This way in the end we can easier interpret the results from PCA as we will have each variable associated to no more than one factor.

We will use PCA to try to answer the following questions:

- What are the main performance bottlenecks?

- What contributes to CPI[20] increase?
- Can we identify the characteristics of a new benchmark?

7.2.1 What are the main performance bottlenecks?

To answer this question we have to answer the following direct questions: how many factors do we extract? What are the factor loadings?

To choose the number of factors to interpret, we can look at the eigenvalues. Each axis has an eigenvalue associated with it, and they are ordered from the highest to the lowest. These values of the eigenvalues are related to the amount of variation explained by the axis. If we notice from the plot that some points tend to level out, this means that we can ignore the components associated to those eigenvalues as those eigenvalues are very close to 0, hence very little of the variance is explained by their associated factors. The elbow criterion[21] helps us choose the number of factors. We look at the plot (Figure 4), and we can say two factors should suffice, however if we interpret the loadings on three factors we have meaningful results with respect to the main performance issues.

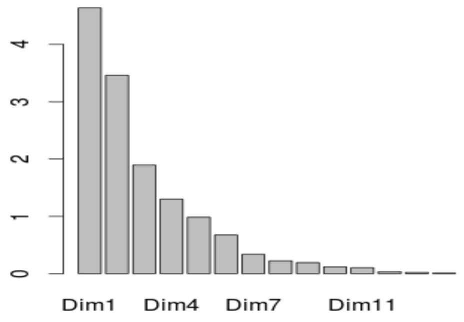

Figure 4: Plot of the eigenvalues

In the following table we show the correlation coefficient between the factors and the performance events (the factor loadings):

PERFORMANCE EVENT	Factor 1	Factor 2	Factor 3
UNHALTED CORE CYCLES	-0.812		0.414
INSTRUCTION RETIRED	-0.819	-0.454	
UOPS RETIRED:ANY	-0.724	-0.610	0.199
UOPS ISSUED:ANY	-0.833	-0.495	0.132
ARITH:CYCLES DIV BUSY		-0.169	**0.936**
ARITH:DIV		-0.177	**0.929**
RESOURCE STALLS:ANY		0.395	0.804
BR INST EXEC:ANY	**-0.884**	-0.199	-0.179
BR MISP RETIRED:ALL BRANCHES	**-0.881**		-0.267
BACLEAR:CLEAR	-0.491		
L2 RQSTS :IFETCH:HIT	-0.423	-0.416	-0.634
L2 RQSTS IFETCH:MISS	-0.869	0.231	

[20] Cycles per Instructions Retired
[21] http://www.enotes.com/topic/Determining_the_number_of_clusters_in_a_data_set#The_Elbow_Method

ITLB MISSES	-0.337		
DTLB LOAD MISSES	0.218	0.703	-0.185
MEM LOAD RETIRED:L1D HIT	-0.131	-0.676	
MEM LOAD RETIRED:L2 HIT	-0.457	**0.805**	
MEM LOAD RETIRED:LLC UNSHARED HIT	-0.138	**0.731**	0.574
MEM LOAD RETIRED:OTHER CORE L2_HIT HITM	-0.164	-0.294	-0.213
MEM UNCORE RETIRED:LOCAL HITM	-0.124	-0.282	-0.207
MEM UNCORE RETIRED:LOCAL DRAM AND REMOTE CACHE HIT		**0.914**	
MEM UNCORE RETIRED:REMOTE DRAM	0.372	-0.334	-0.172
MEM UNCORE RETIRED:REMOTE HITM		-.0282	-0.194
MEM UNCORE RETIRED:OTHER LLC MISS		0.876	

PCA Factor loadings

From the factor loadings we can conclude there are 3 main performance bottlenecks in the benchmarks tested: *Branches, Low level memory access, and arithmetical divisions*.

7.2.2 What contributes to CPI increase?

We plot the vectors corresponding to our variables (performance events) in a unit circle. Using this model we plot CPI (blue) to see where it is displaced in the plot. The cosine of the angle between two vectors is a measure of the correlation between the variables they correspond to.

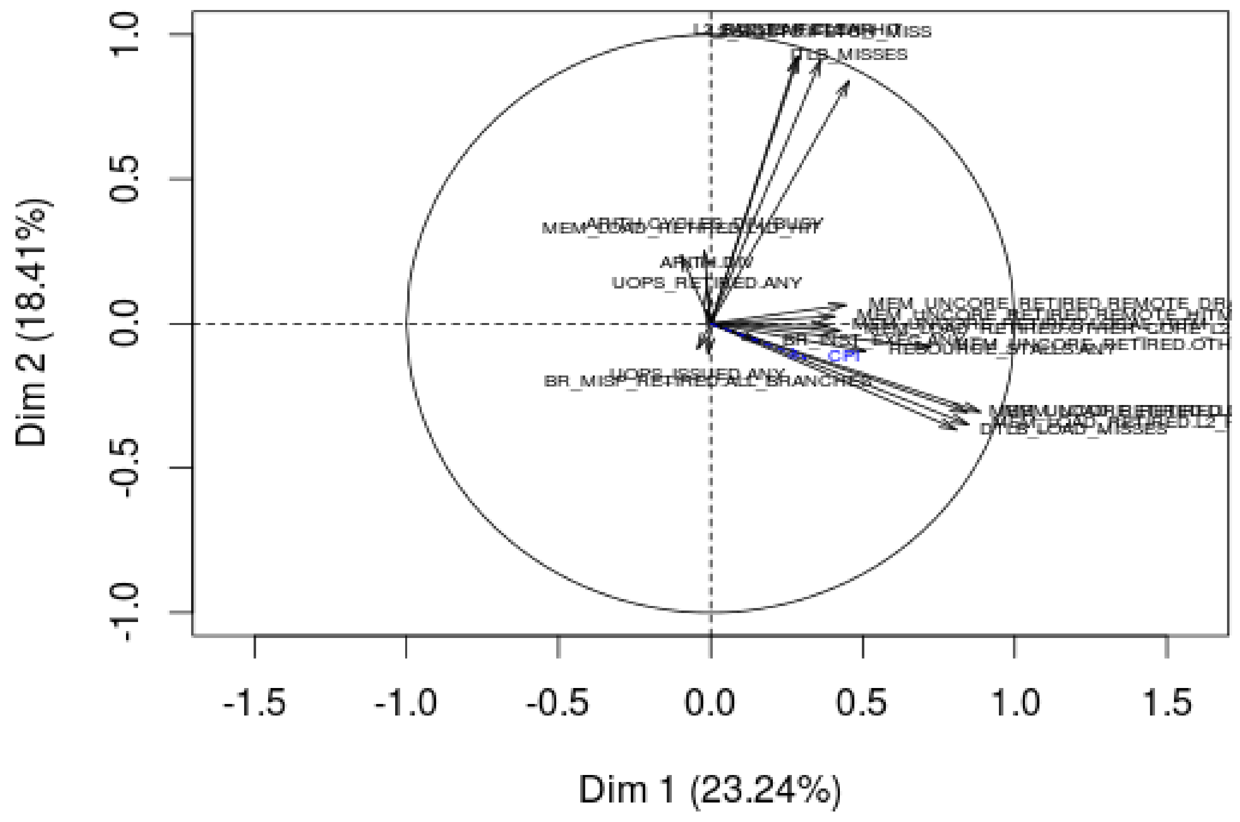

Figure 5: Plot of the variables on the first two axis

When we look at the variables factor map on the first two dimensions where branch events and memory events are well represented (as we noticed by analyzing the factor loadings), we see the CPI is highly correlated with the costly memory events and with the branch events. Therefore, the larger the counts for costly memory events in a benchmark or the larger the counts for branches mispredicted, the larger the CPI for that workload.

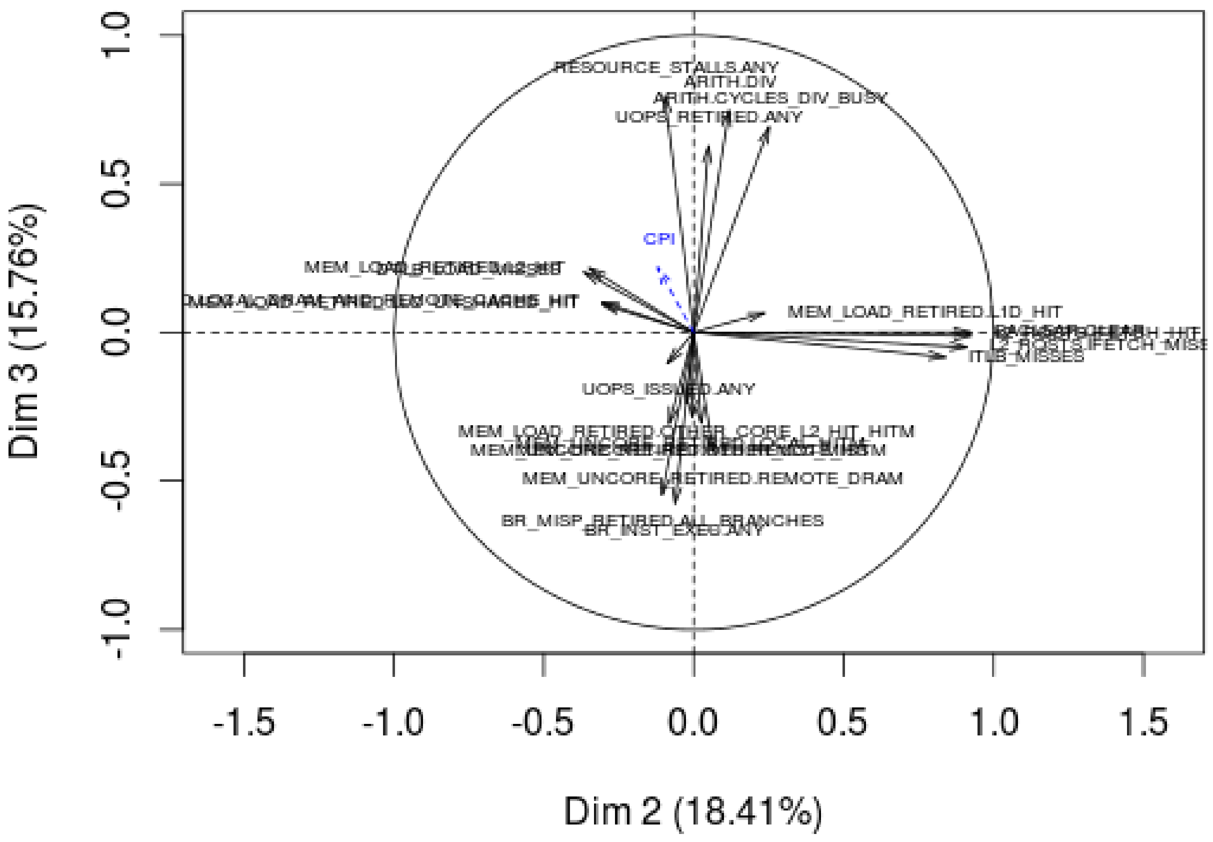

Figure 6: Plot of the variables on the 2nd and 3rd axis

If we plot on the second and on the third dimension, where memory events and arithmetical division events are well represented, we see the angle between the vectors of CPI and ARITH.DIV is small. Consequently, the more arithmetical division operations in the code, the larger the CPI.

Note:

The performance monitoring unit is able to collect counts for a wide range of performance events. In our study we have only used a small subset, of the most common ones, hence our conclusions are limited to the information we can extract from them. The out of order engine is complex and performance bottlenecks can be encountered in various points on the uops flow. Fortunately we are provided with a large number of performance events [8] so we can always involve these techniques with other performance events as explanatory variables to discover new correlations or to assess more obscure ones.

7.2.3 Identification of the characteristics of a new benchmark

In order to identify the characteristics of a new benchmark we can use the model built with all the benchmarks except one we leave out and want to see its performance issues.

We use the idea of biplots to show the explanatory variables together with the dependent variables. For clarity we show them one beside the other but we will interpret them in the same manner as we would interpret the biplot. We left “malloc_test” aside, we did not include it in building the model – it is a supplementary dependent variable, and we just display it based on the knowledge we have from the other benchmarks we used in the model.

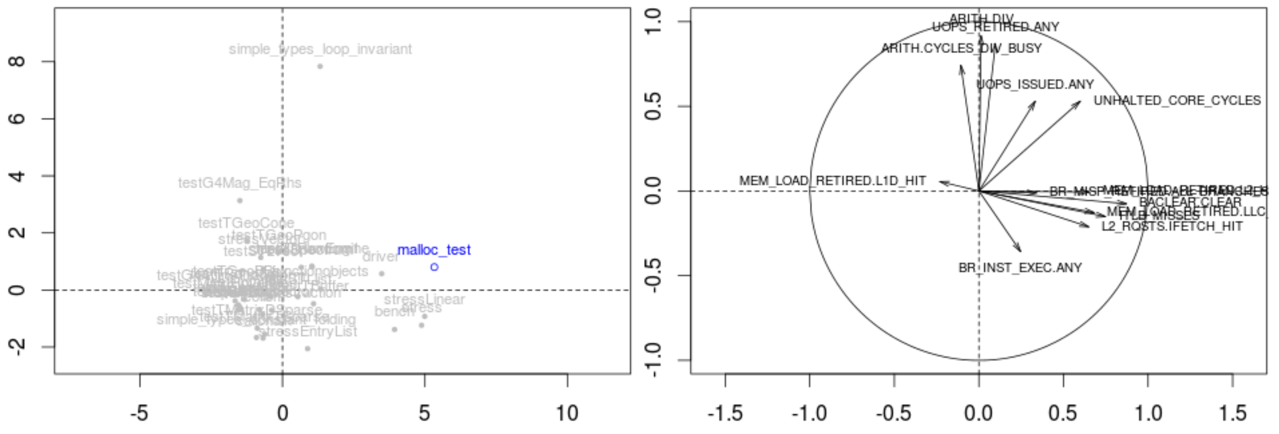

Figure 7: Left - plot of the individuals, Right - plot of the variables

By interpreting the vectors of variables we can say that the first component is highly correlated with costly memory events and the second component is correlated with arithmetical division events.

“Malloc_test” shown on the plot according to its projections, and we see that the model predicts well that it is a memory intensive benchmark, with high counts for costly memory events.

We also see that the second component opposes “simple_types_loop_invariant” benchmark to the “stressEntryList” benchmark the former having high counts in arithmetical division events the latter having very small counts for these.

8 A study on how performance events implicate the choice of compiler flags

We would now like to establish if we can infer the compiler flags that are likely to bring a performance gain for a benchmark, by analyzing the performance events of that benchmark collected from its run at "O2" optimization level. We implement this as a binary classification task.

We collect performance events for each of these tests run without compiler optimization. We also collect the counts for UNHALTED_CORE_CYCLES for the benchmarks run with compiler optimizations on, in order to accurately establish if a flag or a combination of flags brought a performance gain or not.

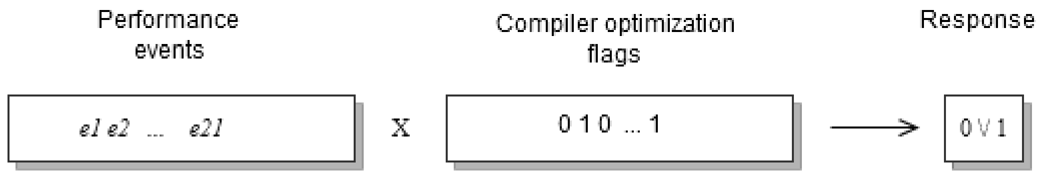

Figure 8: Inference model

We use performance events and compiler optimization flags as explanatory variables. The response is 0 or 1 depending on whether that combination of compiler flags brought a speedup to the benchmark the performance events belong to. We label with 1 the executions that showed at least 1% performance gain with respect to O2. We chose 1% because if we hadn't specified a threshold at all and we labeled as 1 those runs for which less cycles were counted than the execution at O2, the cases where the difference was very small, would have made it difficult to establish which compiler flags are indeed beneficial for that benchmark. If the threshold was higher, then we would have had very few benchmarks for the training set, which is not desirable either. We tried several classification algorithms and we chose the one that dealt best with the problem of unbalanced class distribution, and that had a high precision. The formulas for precision and recall[22]:

$$Precision = \frac{T_p}{T_p + F_p} \quad Recall = \frac{T_p}{T_p + F_n}$$

The reason behind our aim to obtain a good precision at the expense of a lower recall, is quite intuitive. We want a good accuracy as a primary condition. But after achieving this, we don't necessarly want to have in response all the possible combinations that improve performance, but we rather want to be sure that we can rely on those configurations that are labeled "positive" as being good optimizations for the code.

We tried other classification algorithms among which we can enumerate: Random Forests, Support Vector Machines (SVM), Logistic Regression. While SVM with tuned parameters lead to a good

[22] $T_p = True\ positives; F_p = False\ positives; F_n = False\ negatives$

classification accuracy, we obtained better precision with Random Forests. Consequently we will expose only this method and the results.

Note:

- After labeling the 37 benchmarks only 22 of them had at least 1% performance gain with the flags or the combinations of flag we experimented with.
- There are 22 benchmarks in the training set. In total 22 x 786 (configurations) → 17292 observations
- There are 15 benchmarks in the test set

- There are 37 explanatory variables, out of which:
    - 20 continuous consisting of performance events normalized relative to the total number of instructions retired so that we can generalize across benchmarks
    - 17 categorical → the compiler optimization flags
- Unbalanced class distribution[23], only 24% of the cases are labeled with 1

Random Forests is a machine learning technique that can be applied for both regression and classification tasks. Classical methods don't work or have problems in building a model when the number of explanatory variables is greater than the number of observations. Random forests can deal well with such data sets and produce good results. In Random Forests we have a set of classification trees. We do not use cross-validation as a method of asserting the quality of the model but the out of bag error estimate (OOB). OOB can give us an unbiased estimation of the error by constructing each tree with a different bootstrap sample from the data. For a tree from the forest, let's say *Tj*, approximately one third of the observations are left out the sample and used as test set for it. Doing this for each tree, we in the end have for each observation, about one third of the trees assessing a label for that observation. The majoritary label is the one that is finally returned for that observation.

For building a Random Forests model we used the "party" package from R, where cforest() function represents the implementation of the random forests algorithm. There is also an implementation in "randomForest" package but this one is better it attenuates the bias that random forests have towards highly correlated variables[24].

The parameters we will tune are:

- ntree → the number of trees in the model
- mtry → the number of randomly preselected variables
- mincriterion → the depth of the trees which is eventually left to the default value

[23] For a bi-class classification problem we deal with unbalanced class distribution when we have a larger number of cases falling in one category than in the other.

[24] C. Strobl, J. Malley, and G. Tutz. An introduction to recursive partitioning: Rational, application and cjaracteristics of classification and regression trees,bagging and random forests. Psychological Methods,14(4):323-348, 2009

For ntree=150, mtry=37 (which is equal to the number of candidate predictor variables) we obtained the most satisfactory results.

The Out of bag cross-classification of true vs. predicted classes gives us the following numbers:

Accuracy	**Precision**
92%	82%

These numbers are reported for the OOB cross classification on the 22 benchmarks used for the model. We also test the model on the benchmarks for which none of the flags brought a performance gain of at least 1% relative to the execution time when compiled with –O2. We label as 1 those executions that have a number of cycles smaller than the number of cycles used after compiling with –O2 (despite the fact that this difference is very small). For these cases the precision varies largely between 20% and 77%. This is because that labeling is not accurate and the small differences are not necessary caused by the compiler optimization flags. Hence, comparing the predicted labels with the true labels (established by the method mentioned above) is not a good measure for evaluating the algorithm.

We will also try to establish the importance of the predictor variables in building the model, since Random Forests provide means for establishing variable importance. The technique is the following: each tree, after it is built, takes the OOB data and they start randomly permuting the values of each variable. For example for variable V, the values are permuted and then the trees assess the labels of the OOB with the permuted variable V. We compare these labels with the labels assigned on the OOB set without permuting values in variable V. Intuitively, if there is no significant difference it means that that variable has little significance in deciding the label for the cases. So, the trees take the difference of the correctly labeled observations from the two sets (clean OOB and with variable V's values permuted) and they average this on all the trees. This value will be the raw importance score for variable V.

Establishing the importance of the predictor variables is meaningful as these variables can be compared with respect to their impact in predicting the response or even their causal effect. One interpretation on variable importance would be how much more useful than random a particular explanatory variable is in successfully classifying data. Quantification of contributions allows the prioritization of performance issues and guides workload tuning.

The results for variable importance measurement are shown in the figure bellow:

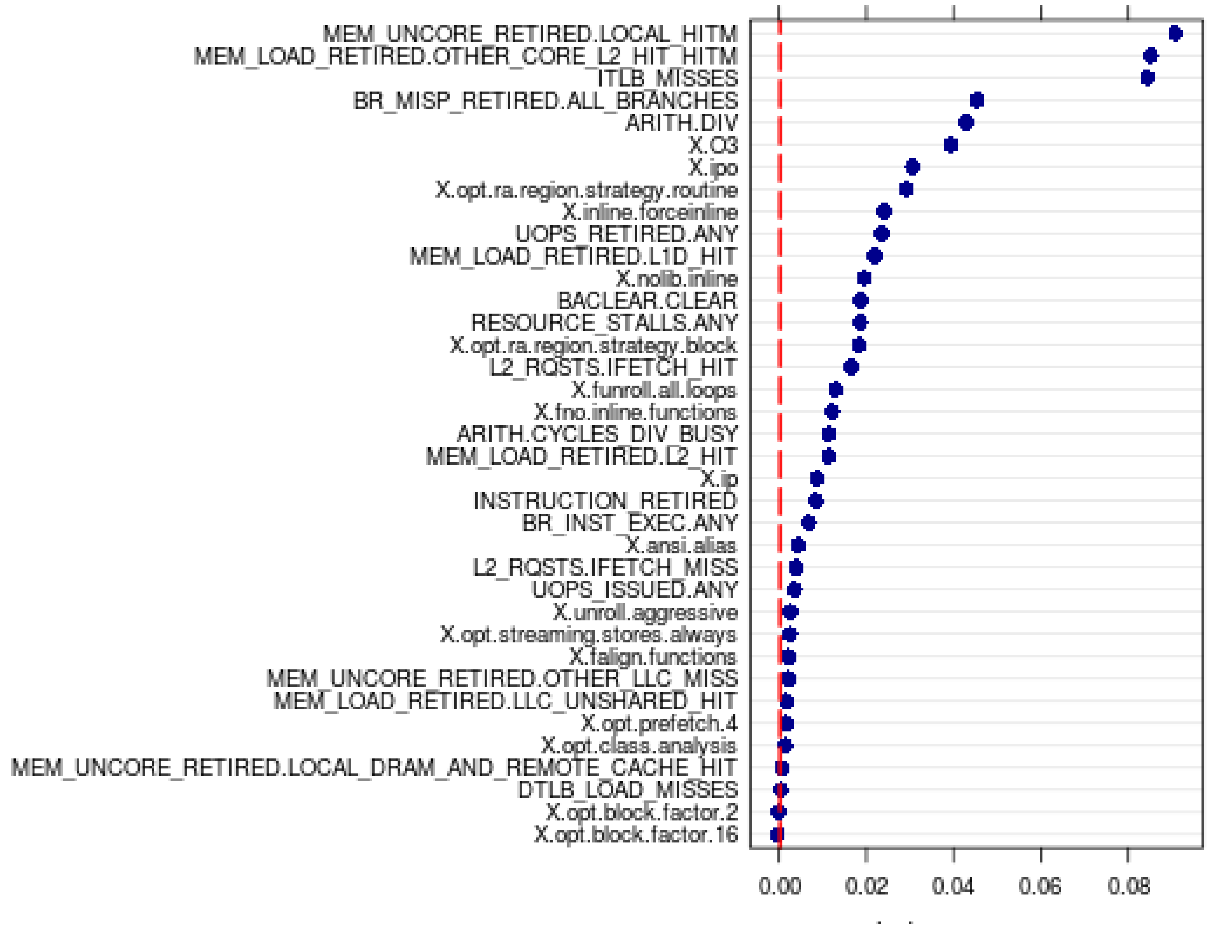

Figure 9:Variable importance in the dataset (predictors to the right of the dashed vertical line are significant)

The further a point is situated relative to the red line, the more influential the variable is. This way we can see that events that count ITLB misses, low level cache hits, arithmetical division or branches misspredicted are very weighty for building the model. According to the model, the most powerful compiler optimization flags are O3, ipo, opt-ra-region-strategy=routine and inline-forceinline.

9 Conclusions

Performance analysis is a challenging and appealing task. At times, identifying the key issues can be straightforward but most often one has to perform exhaustive investigations.

Machine learning is now widely used in a vast number of areas. Through our study we showed how data mining and machine learning can produce interesting results also in the field of performance tuning. Performance events are the most accurate information one can get about the execution of the application (without employing code instrumentation). As the collected numbers are specific to a given code, we wanted to go to a more abstract level and find patterns that are applicable on a larger scale.

We have built a model that is able to associate performance bottlenecks with the compiler optimization flags that are likely to attenuate them, yet there is always space for further development. Our random forests model can receive as input performance events collected from a benchmark compiled at "-O2" optimization level and output compiler flags or combinations of compiler flags that would be suited for improving runtime performance.

The projection matrix obtained after performing Principal Component Analysis can also be used to obtain insightful information for new cases we don't know anything about. For example we can collect performance events from a new benchmark and use the PCA model to have some insight about what the performance issues of that benchmark could be.

In order to make our models more feasible and more stable we could increase the size of our training data by selecting benchmarks that are more distinctive, addressing specific performance issues. Machine learning is a far-reaching domain and the more data we feed to the machine learning algorithms, the clearer their potential is.

10 References

[1] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. *Rapidly selecting good compiler optimizations using performance counters*. International Symposium on Code Generation and Optimization, 2007

[2] H. Dong and V. Jeffrey. *Scalable analysis techniques for microprocessor performance counter metrics*. Proceedings of the IEEE/ACM SC, 2002.

[3] S. Bird, A. Phansalkar, K. Lizy, A. Mericas, and R. Indukuru. *Performance characterization of SPEC CPU benchmarks on Intel's core microarchitecture based processor*. SPEC Benchmark Workshop, January 21, 2007.

[4] M. Stockman, M. Awad, R. Khanna, L. Christian, H. David, E. Gorbatov, and U. Hanebutte. *A novel approach to memory power estimation using machine learning*. ICEAC, 2010.

[5] G. Contreras and M. Martonosi. *Power prediction for intel xscale processors using performance monitoring unit events.* Low Power Electronics and Design, 2005. Proceedings of the 2005 International Symposium, 2005.

[6] Y. Wucherl, K. Larson, L. Baugh, K. Sangkyum, A. Wonsun, and R. Campbell. *Automated fingerprinting of performance pathologies using performancemonitoring units*. International Conference on Measurement and Modeling of Computer Systems, London, 2012.

[7] Cemal Yilmaz. *Using hardware performance counters for fault localization.* Proceedings of the 2010 Second International Conference in Advances in System Testing and Validation Lifecycle, 2010.

[8] David Levinthal. Performance *analysis guide for intel core i7 processor and intel xeon 5500 processors.* Proceedings of the sixth ACM workshop on Scalable trusted computing, 2008-2009.

[9] Intel Software Development Tools. Intel C++ compiler user and reference guides

[10] David W. Shrewsbury. *Reducing the impact of software prefetching on register pressure.* Proceedings of the 2000 ACM symposium on Applied computing - Volume 2, pages 767-773