Inlining Assistance for Large-Scale Object-Oriented Software

Technical documentation

Department: Information and Communications Technology
Field: Computer Science
Student: Baptiste Wicht
Supervisors: Paolo Calafiura, Roberto Agostino Vitillo
Professors: Frédéric Bapst, Ottar Johnsen
Expert: Noé Lutz
Project Period: July 31, 2011 - August 12, 2011
Date: August 11, 2011
Version: Version 1.0

Preface

This thesis is submitted in partial fulfillment of the requirements for Bachelor of science in Computer Science at the University of Applied Science, Fribourg for Baptiste Wicht. This project took place in the Lawrence Berkeley National Laboratory, California during an eleven weeks period.

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 for this very interesting project and for the opportunity to do this project in LBNL
- Roberto Agostino Vitillo for his support and precious advices throughout the project
- Frédéric Bapst for his help and the hours of Skype meeting that he devoted me
- The LBNL software development team for the good moments passed together
- Ottar Johnsen and Noé Lutz for their help and for coming to visit us at LBNL
- Sara Moschetto for her excellent proofreading of this document

Abstract

The goal of the project is to create a tool that assists the developer in his task of interpreting a call graph to reduce the performance impact of function calls in large-scale object oriented applications. The main way to reduce this overhead is to inline some functions or call sites in the application.

Inlining is an optimization that replaces a function call with the body of the called function. As inlining brings both performance benefits and additional costs, the difficulty is to detect where it is advantageous to apply it. Compilers have their own heuristic to perform automatic inlining, but if we want to go further, we must take more parameters into account.

During this project we studied in details the function invocation process and how inlining can influence a program's execution time to develop a temperature heuristic aimed at detecting which functions and call sites are the most interesting to inline. This heuristic is applied to a complete call graph of a program's execution. The analyzer is also able to detect clusters of hot call sites related to each other. We also studied the overhead of virtual function calls and invocations through libraries. The application can detect broken virtual hierarchies and very expensive library dependencies.

This project resulted in an analyzer for performance issues that can be used from command line or as a library in another application. This tool allowed us to improve the performance of well-known application.

Baptiste Wicht
Apprenticeship as Computer Scientist
Computer Science Bachelor degree
Switzerland HES-SO University of applied science

Contents

1 Introduction

This document describes the Bachelor of Science thesis of Baptiste Wicht on a software tool developed to improve the ATLAS experiment performance.

1.1 Context

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). 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 that registers 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.

The basic software abstractions and tools for HEP are provided by the GAUDI[1] framework. On top of this framework, the Athena framework provides the ATLAS-specific software interfaces and services. Using the GAUDI terminology, a typical HEP analysis application used for the evaluation of the performance of physics objects is made of algorithms that read data objects from a transient data store, such as reconstructed jets, and output statistical data in the form of histograms.

This project is a part of a collaboration of the Lawrence Berkeley National Lab (LBNL) ATLAS group with Google that aims to develop a software for performance analysis based on the Linux perf events system. The main goal is to provide a tool to analyze the call graph of an execution of ATLAS software, find issues and propose solutions to the user in order to improve the overall performance. The detected issues will be of different types, like:

- functions to be inlined
- functions that should be in the same library
- clusters of hot functions that are to be inlined together.

Indeed, the cost of invoking a function is not negligible. Before invoking a function, the context must be saved, the parameters must be pushed to the stack (and sometimes their values must be copied), then the function can be executed. After that, the return value (if there is one) is retrieved from the stack and the caller context is restored. These operations are specific to an architecture and are described in detail in an Application Binary Interface (ABI). The ATLAS software is mostly written in C++ compiled with GCC[2] and executed on GNU / Linux platform, so the research will focus on these technologies. Some

parameters can prevent a function to be inlined directly by the compiler. In most cases, if a function is a good inlining candidate, but has not been inlined by the compiler, it is an issue that the developer would like to be aware of.

The compiler cannot solve every problem concerning inlining. Indeed, it has only a part of the information. It has a limited view on the whole application because most of the files are compiled one by one. The goal is to have a better point of view of the whole application to make decisions about inlining. Moreover, it only has static information about the application, where it generally more efficient to optimize based on the runtime information of the application.

The problem of choosing the most profitable call-sites to inline while minimizing the associated costs is known as the *Substitution Problem*. According to R. Scheifler, we can give an efficient reduction of the Knapsack Problem to this problem to show that the Substitution Problem is at least as hard as the Knapsack problem[3]. This last problem is known to be NP-Complete, so solution to our problem will require exponential time to be found. That is why we will use a heuristic to estimate the interest of inlining a given function or call site.

1.2 Technical Information

As said in Section 2.4, this tool is part of a bigger tool chain. It will be invoked by another tool, responsible for the perf events gathering. Then, the output of the tool will be passed to the visualizer. Most likely, the visualizer will be a web application written mostly in JavaScript. Our application will run on the server side as a C++ program.

This application will receive a call graph generated by either Callgrind[4] or Gooda (Google Optimizations Data Analyzer), the perf events data analyzer. The application must be designed in a way that allows us to switch between these two call graph generators. The input will be a DOT graph file. The graph manipulation library has not been chosen yet, so a priority will be to choose one. Once a call graph has been received, the application has to run the analysis on the complete graph and annotate it with the information of the analysis as well as the information of the feature extraction phase. The application will output its issues and solutions in the console as a command-line tool. It is possible that the project will later be included directly in Gooda in which case the output will be the annotated call graph and the list of issues, in JSon format in order to be able to parse it from the visualizer.

1.3 Goals

Based on the above, we have to make some approximations to find a good solution. The first objective of the project is to provide a very basic heuristic function to determine the interest of inlining a particular function or a specific call site. We have named this heuristic value the *temperature*. The hotter the objects are the more interesting they are to be inlined. To start, this temperature function will consider the cost of a call site and the cost of the callee body. This cost must fit with the reality and must be proportional to the real cost of invoking a specific function. The cost of invoking a function depends on the number of instructions needed to put the parameters on the stack, to save the state of the registers,

to restore it and to retrieve the return value of the executed function. Then, in order to improve the precision of the temperature, we must take into account, not only the function itself, but also information about the caller and the associated stack trace. The function will not be exactly the same for a function or a call site because we do not have exactly the same information, but the general idea is the same.

Another objective is to detect some specific patterns in the call graph. In this context, a pattern can be of several types: a kind of group of hot call sites related to each other (cluster) or a group of functions in different libraries that are frequently invoking each other. At this time, it is not clearly defined how to detect them and what to do with them once they are found. The idea of detecting clusters is an idea that came during the project's first phases. It is not a common technique, so nothing is proven. It is possible that it will not be useful in terms of performance. To the contrary, it may help identify some big issues in the project. The idea is to consider first clusters of call sites as inline candidates before considering single call sites.

After proposing a first version of the temperature function, it will be interesting to improve it by considering more features to find new issues and be more reliable. Section 2.4 gives more details about the features that may be used to improve the temperature heuristic.

1.4 Structure of this document

This document is divided into eleven chapters, including this introduction. Several appendices are also available at the end of the document.

The second chapter discusses the analysis made during this project about several subjects. Then, the next chapters are presenting the main phases of the implementation: the information extraction and the information analysis. The fifth chapter presents the design that has been chosen for the application. The next section talks about the tests made for this project. After these chapters, an analysis of the performances of the application is presented. Chapter eight and nine present refactorings made during the project in order to improve the application and the problems encountered during the implementation. The tools used during this project are then briefly presented.

Finally, the report is concluded with a presentation of the project's results. This section also presents the techniques learned during this project. Moreover, this last chapter also lists some work that may be done to improve the application.

After the main chapters, you will find two appendices, one describing the content of the CD-Rom coming with this report and the other one giving details about an analysis made during the tests. Then, you will find the list of references used during this project, a glossary and an index and the lists of tables, figures and listings present in this document.

2 Analysis

2.1 Callgrind

As said in Section 1.2, the call graph can be generated by either Callgrind or Gooda. At this point of the project, Gooda has not been released and it cannot be used, so Callgrind will be used during this project until Gooda can be used. The information contained in these two call graphs is almost identical and both can be converted to a DOT file.

It must be simple to change from Callgrind to Gooda. Either by automatically identifying the type of input file or by providing command-line options to specify the input format.

Callgrind is a tool from the Valgrind[5] project. Valgrind works by using a synthetic processor. Your executable is read and converted into an intermediate representation that is ran by the Valgrind virtual processor. Then, the launched tool instruments your program adding some more code to get information during the run. The performance slowdown of Valgrind can be very high, to an order of magnitude of 10 to 50 depending of the profiled application.

This tool records all the function calls made during a program's execution and creates a call graph representing all the calls. In the graph, a vertex represents a function and an edge represents a call site. More than the relationships between the different functions of the program, this tool also records the cost of each function and call site. A function has two costs:

- the self cost: Only the cost of the instructions belonging the function
- the inclusive cost: The sum of the function self cost function and the inclusive costs of every called function

As Callgrind runs in a virtual processor, the cost is only the number of instructions fetched by the function because Callgrind is not meant to calculate a real time. So, we do not have real information about how long a function takes because it is not representative of a real processor.

The Callgrind file output is a flat file with every function information and call among functions. Names are compressed and information is grouped. This file is not made to be humanly readable. We can visualize information from a Callgrind file in a user-friendly way using KCacheGrind[6]. This tool groups all the information per function so it is convenient to see its callers, callees and costs. It is very useful to view all the data contained in a Callgrind profile data file. KCacheGrind provides also partial view of the call graph.

Because our application will receive the call graph in a DOT file, we needed a way to convert a flat Callgrind file to a DOT graph file. The tool used to perform this job is Gprof2Dot[7]. This tool is a simple Python script that takes a Callgrind file and converts it to a DOT graph. This tool is intended for visualization tools, so there is a lot of information in the DOT file like colors, sizes, fonts, distance, etc. that are not useful for this project, but they are useful if we want to visualize the graph, for example with the dot[8] tool of the Graphviz[9] distribution. This tool converts a DOT file into a PNG image showing the graph.

Only Callgrind and Gprof2Dot are essential to the project because they are generating the call graph files, but the other tools are also very helpful in order to understand the call graph at its different steps.

2.1.1 Usage

Every tool in the Callgrind tool chain is a command-line tool.

Like Listing 1 shows, you have to profile your application using Callgrind.

Listing 1: Generate Callgrind profile

```
valgrind --tool=callgrind --demangle=no --compress-strings=no --compress-pos=no
   program [program_options]
```

The result will be stored in a *callgrind.out.XXX* file where XXX will be the process identifier. Then, you have to convert the Callgrind file into a DOT file. Listing 2 shows the necessary commands for this step.

Listing 2: Convert Callgrind file to a .dot file

```
python converter.py -f callgrind -e 0 -n 0 callgrind_file > callgrind.dot
```

This operation can take a very long time depending on the size of the call graph.

If you want to visualize your call graph, you can convert it to a PNG image using the dot tool as presented by Listing 3.

Listing 3: Convert a .dot file to a PNG image

```
dot -Tpng -o graph.png callgrind.dot
```

2.2 Gooda

Gooda (Google Optimization Data Analyzer) is a Google project aiming to provide a better view on the perf events. It analyzes the data obtained from perf tool. The perf tool is a Linux kernel-based framework. This tool provides, besides other features, a profiler. This profiler records hardware and software events like CPU cycles, cache-misses, page faults, etc. The perf system mainly uses hardware counters to get the information from the system without slowing it down too much. Modern CPUs have some hardware registers to count, for example, the number of instructions executed, the number of caches misses or the misprediction of branches. Perf reads these registers to profile the application. The advantage of perf over tools like Callgrind is that there is much less overhead due to profiling and more pieces of information. Contrary to general profilers, not all events are available in all CPUs. So, profiling an application in two different platform can produce different events. Of course, due to the sampling, we have only the main information, so we can lose some precision compared to Callgrind. However, this precision loss is generally only impacting non frequent functions.

The perf output file (perf.data by default) is not human-readable. It is a binary file containing the information about every sample. At the present time, the file is not compressed and can grow very quickly. It is not comparable to a Callgrind profile file because a perf file contains a lot more information about each event that occurred during the profiling period, not only about the function calls.

Perf can also generate a call graph. But it was not used during this project. Indeed, perf generates the call graph in standard output, in a format meant to be read by human, although this is not really the case. It is not easy to parse it. Moreover, in this call graph we do not have the number of calls of each call sites, we can only see the percentage of time spent on each call sites of the current function. Furthermore, the least costly functions are excluded from the call graph so that in our case, we can lose some interesting information.

Gooda analyzes the data generated by perf and generates spreadsheets of the results. More than spreadsheets, Gooda also generates a call graph of the given data. Gooda has also to parse the libraries to obtain information about the source code, the basic blocks and the functions in order to generate the spreadsheets. Gooda manages several views: the source view, the basic block view and the assembly view.

At the beginning of this project, it was not possible to have access to Gooda, because there was no NDA between Google and LBNL. Once the NDA has been signed, it has been possible to make some tests using the tool. Nevertheless, the call graph generation feature of Gooda has not been finished before the end of the project, so we have not tested our tool on the call graph generated by Gooda.

2.2.1 Usage

If perf has been packaged for your system, you can install it using your package manager. Listing 4 shows how to make this installation using apt-get.

Listing 4: Install perf

```
sudo apt-get install linux-tools-2.6.38-8
```

Like almost every Linux tool, perf is a command-line utility. If you want to record perf events, you have to use the *perf record* command, as shown in Listing 5.

Listing 5: Record perf events

```
perf record your_application [your_application_options]
```

perf also provides tools to visualize the data contained in the report, like perf report in Listing 6.

Listing 6: Perf report of your profile

```
perf report
```

This command-line tool will list every function of your application, with the function cost, the command used as well as the shared object containing the function. Listing 7 shows an example of a perf report.

Listing 7: Example of a perf report

```
# Events: 561  cycles
#
# Overhead        Command             Shared Object
# ........  ............  ........................  ........................
#
    23.22%       inlining  libstdc++.so.6.0.14       [.] 0x9cff4
    13.26%       inlining  inlining                  [.] 0x1509a
    11.95%        readelf  libc-2.13.so             [.] vfprintf
     6.07%             sh  [vesafb]                 [k] 0xffffffff8103804a
     4.50%       inlining  libc-2.13.so             [.] _int_malloc
     3.42%       inlining  [vesafb]                 [k] 0xffffffff8103804a
     3.17%        readelf  [vesafb]                 [k] 0xffffffff8103804a
     2.92%        readelf  libc-2.13.so             [.] _IO_new_file_xsputn
     2.59%        readelf  libc-2.13.so             [.] ___printf_chk
```

We can also generate the call graph with perf report, as shown in Listing 8.

Listing 8: Perf call graph

```
perf report -g
```

The call graph is then printed in the console (you can navigate in the view like in the *more* command-line utility). Listing 9 shows an example of such a result.

Listing 9: Example of a perf report call graph

```
# Events: 3M cycles
#
# Overhead         Command             Shared Object
# ........  ..............  ........................  ..................
#
    76.13%        inlining  libstdc++.so.6.0.14       [.] 0x9ce00
            --- 0x7fc27242cb04
                Infos::parseFile(std::string)
                Infos::sizeOf(std::string, std::string const&)
                GraphReader::parseVertices(CallGraph&)
                GraphReader::extractInformation(CallGraph&)
                GraphReader::read(std::string)
            --- 0x7fc27242d00b
               |--82.30%-- Infos::parseFile(std::string)
               |          ...
               |--5.25%-- void std::_Destroy<std::string*>(std::string*, ...
```

Again, for each function we can find the command and the shared object related to it. For each end function, we saw the stack trace.

2.3 Function call

Invoking a function is a complex operation and the detailed mechanism strongly depends on the compiler and particularly the architecture. In this section, we will see how a function is invoked.

The exact way of invoking a function is described in an Application Binary Interface (ABI). A compiler uses this ABI specification to generate a code that matches the current architecture.

There are two main kinds of function call convention, depending on which of the callee or the caller remove the arguments from the stack:

- __cdecl: This convention supports variadic function, so the caller must clean up the stack after the function call because the called function has no way to do this. Indeed, the called function can be called with any number of parameters, so that it is not possible here to know at compile time how many parameters will be passed.
- __stdcall: Each function needs to take an announced number of parameters, so the called function can do the argument cleanup itself. This convention is mainly used in the Win32 API. The advantage of this convention is that the code to clean up the stack is present only once for each function which can result in a significantly smaller application.

This project is intended for the GNU/LINUX world, so this analysis will focus on the __cdecl convention and on the gcc compiler.

First of all, the arguments have to be passed to the callee. Generally, they are passed on the stack, but a compiler can optimize this process using registers. The number of arguments that can be passed using registers depends on the architecture. For example, on x86_64 architecture, you can pass up to six parameters using registers, and on ARM architecture you can pass up to four parameters in registers. Some arguments cannot be passed in registers if they are too big and some elements have to be copied before they are passed to the callee.

When a non-static member function is invoked, the *this* pointer is passed onto the stack last, as if it was the first argument of the function.

Then, the old instruction pointer (pointing to the first instruction after the *call* instruction) is pushed onto the stack. It will be used as the return address. This is generally done automatically by the *call* instruction.

Now we are in the callee body. The old base pointer is pushed onto the stack and the base pointer is set to the value of the current stack pointer. The base pointer is an utility pointer to refer to the start of the function, so we can easily address the parameters and the local variables using this pointer. In some architectures, this can be achieved using the *enter* instruction, but not many compilers use this

instruction. Indeed, it consumes more cycles than the equivalent instructions (15 clocks instead of 6) although it is more compact (6 bytes instead of 9).

The callee has to save the registers it will use. For every modified register, it has to save the old value onto the stack. Of course, the registers used to store parameters do not have to be saved.

The function can now allocate space on the stack to save local variables. Again, the *enter* instruction can automatically allocate space on the stack for the local variables. Generally, the compiler tries to put as many variables as possible directly in registers. But if there are too many variables, the compiler will put them onto the stack.

Figure 1 shows the state of the stack if everything is passed onto the stack.

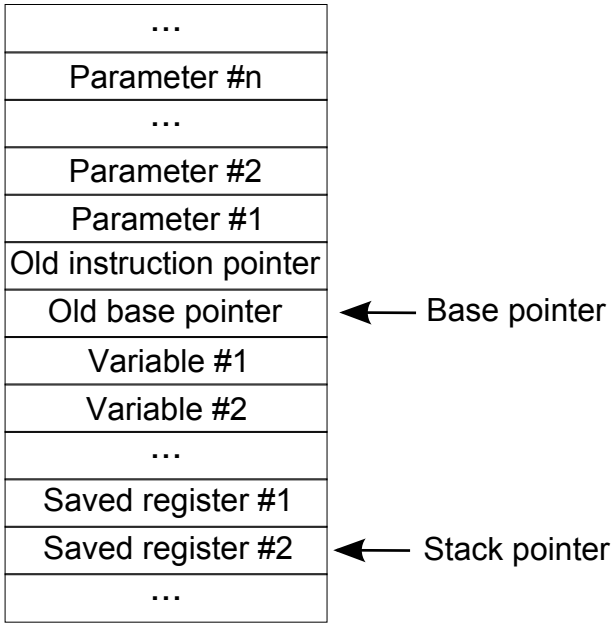

Figure 1: Stack during function call

Once everything is allocated, the body of the function is executed.

The return value is passed into the *%eax* register, so the callee has to fill the register and the caller can read it. If the value does not fit in a register, it is passed on memory with *%eax* indicating the address of the value. Some compilers may use two registers simultaneously to return a value fitting into two registers (gcc use *%eax:%edx* to store such return value). Depending on the return value type, it will perhaps be copied before passing it to the caller.

Then, the called function has to reverse the whole process in order to clean the stack. So it will release the stack variables by incrementing the stack pointer for each variable. Then, for each saved register, it will restore its old value. The last thing the function has to restore is the old base pointer in the current base pointer. To *leave* instruction can be used to restore the base pointer. Once the callee has restored everything, it can return using the *ret* instruction. This instruction automatically jumps to the saved

instruction pointer and removes it from the stack.

Finally, the caller must clean up the parameters, incrementing the stack pointer with the size of the parameters on stack (it is possible that there is no parameter on the stack). And then, the caller can get the return values from register or from the memory.

The process described before applies for every function call, but in some cases, there are differences that can influence the performances of the call.

Non-virtual functions are resolved at compile-time. The called function's address is directly known when invoking it. But virtual functions are resolved only at runtime because the compiler does not know of the object type. In the case of gcc, the virtual functions are resolved using a so-called v-table (virtual table). Each class with at least one virtual function has a v-table. A v-table contains pointers to each virtual function of the class. When a new derived class is created, a new v-table is created for the new class with the address of the member functions to invoke. Each object has a pointer to its corresponding v-table. When a virtual function is called, the processor has to read the pointer to the table and then load the function address from the v-table and finally call the function. We have two additional operations than in the case of a non-virtual function call. In the case of multiple inheritance, the derived object has one virtual pointer for each virtual table. The function call itself does not add runtime overhead in this case, but will need adjustment of the *this* pointer. If we have virtual inheritance, there is also a small overhead due to the calculation at runtime of the base class offsets. Moreover, the virtual functions have another cost, they can generally not be inlined.

When the callee is in another library than the caller, there is also a small overhead. Indeed, when the application is linked with a shared library, the function call is not made directly to the library because we do not know where the library will be loaded in the system. At runtime, for every function call to a library function, you have to first search the function address in the library function table and the then call this address. This is no the case for a static library because the code of the static library is included in the executable at link time with no runtime overhead.

To summarize, here are the main parameters that have a role in the cost of invoking a function:

- The number of arguments (considering that a member function has always one more parameter)
- The fact that the arguments are passed by value or by reference
- The size of the arguments
- The number of registers the function has to save
- The return type of the function if it is not void and the size of this type
- The virtuality of the function
- The fact that the callee is in another library than the caller

There are others parameters that can increase the cost of the invocation, but these parameters are the one that depends of the developer.

2.4 Inlining

Inlining is an optimization that replaces a function call with the body of the function itself. The benefit of this technique is to reduce the overhead of the call. Nevertheless, inlining has several drawbacks. In this chapter, the effects of inlining on an application are studied.

We will start by studying the advantages of inlining. As stated before, inlining reduces the call overhead. As said in Section 2.3, invoking a function requires several instructions and the number of instructions depends on several parameters. When the function call is replaced with the body of the function, all these instructions are saved, generally resulting in an improvement of performance.

Moreover, when a function body is inlined in the caller, the compiler can optimize the call and the callee body at the same time. This may result in new optimizations. Indeed, the compiler may find some conditional branches that are always true or false and this may also enable the elimination of some dead code, the motion of loop-invariant code outside a loop, or even the elimination of some variables and other optimization depending on the capabilities of the compiler. Once again these optimizations are not guaranteed. It is quite possible that for a specific function there is nothing more to optimize.

Let's take a very simple program as an example:

Listing 10: Base function

```
class Dummy {
  public:
    virtual int compute(int value);
};

int Dummy::compute(int value){
  return value == 0 ? 2 : value * value;
}

int computeDummy(Dummy& dummy){
  return dummy.compute(0) + compute(2) + compute(4);
}
```

It is a simple example, but it is interesting in the context of inlining. In the function *computeDummy*, the compiler cannot inline the function *compute* because it is virtual and can be called with several different classes with different implementations. But if this function is always called with an object of class *Dummy*, all calls to the *compute* function can be inlined, so it will results in a code like this one:

Listing 11: Inlined function

```
int computeDummy(Dummy& dummy){
  return (0 == 0 ? 2 : 0 * 0) + (2 == 0 ? 2 : 2 * 2) + (4 == 0 ? 2 : 4 * 4);
}
```

Now several things are known at compile time. Every condition is known at compile time and the compiler will directly replace them with the good branch of the condition. Once the condition has been

replaced by the body of the good branch, only a simple calculation (2 + 4 + 16) remains and once again the compiler can optimize this code by replacing the computation with its result:

Listing 12: Optimized function

```
int computeDummy(Dummy& dummy){
  return 22;
}
```

In this particular case, inlining is very interesting. Three function calls, three conditions and a computation with three additions have been avoided. Of course, this is an optimal case and we will certainly not find an issue like this one in most of the analyzed programs.

Another advantage of inlining concerns the pipelining of modern CPUs. These CPUs use instruction pipelines to optimize the code execution at runtime. A function may break the pipeline when it is invoked and again when it returns. With inlining, we no longer have the problem because there is no call to break the pipeline. The CPU tries to predict the branch taken after an instruction so that it will not break the pipeline, so in some cases, the pipeline is not broken when a function call occurs. Of course, once the function call is replaced with the body of the function, we can have more instructions breaking the pipeline.

To the contrary, inlining has also several drawbacks. First, it increases, sometimes dramatically, the code size of the application. It is especially a problem in embedded systems. It is not reasonable to inline every function. This code size growth also has an impact in the compilation time because the more code there are to compile, the more time it takes.

This increase may also cause Instruction Cache misses. A miss occurs when the instruction we want to fetch is not in the Instruction Cache and the processor has to load it from the memory and copy it in the Instruction Cache replacing another cache line, causing much delay. When a call site are inlined, some instructions are duplicated, so the same group of instructions is at several different locations, increasing the possibility of cache miss because executing the same function now needs several times more Instruction Cache loads. It may cause a critical section of the code to not fit in the cache, decreasing the speed of the whole application. Generally, it is better to optimize a code for Instruction Cache instead of optimizing for reducing overhead of call because Instruction Cache misses cause more overhead than a simple call.

Moreover, adding new code in an existing function may mean adding new variables. Generally, the compiler tries to bind as many variables as possible directly to registers. If the number of variables exceeds the number of hardware registers, a state called register pressure, the compiler has to put some variables in RAM. Having variables in memory means that for every operation on this variable, there is at least one operation in RAM. This process is called spilling. The memory access time is higher than those of the registers, resulting in, sometimes significant, performance losses.

To summarize, here is the list of the advantages and disadvantages of inlining a function call:

- + no more call overhead
- + possible improved optimizations
- + pipelining improvements
- - code size growth
- - compilation time growth
- - Instruction Cache misses
- - Register pressure

2.5 Compiler inlining

Because this project is made for the Linux platform, this section talks about the gcc compiler. Some of the information presented here will be very similar to other compilers, but there are some differences about the strategies of inlining that the compilers use.

If you want to tell GCC to inline a function, you have to add the inline keyword in front of it. By default, GCC will consider any member functions defined within the body of the class as inline (you can use -fno-default-inline to change this behavior).

Telling GCC that a function is inline does not mean that GCC will inline it, it is a way to indicate that this function is a candidate for inline substitution.

Even if the function is marked inline, GCC considers a function as unsuitable for inline substitution[10] when the function:

- is a variadic function
- uses sized data types
- uses the alloca instruction: allocation of memory on the stack
- uses computed goto: a goto to value based on labels, not directly a single label
- uses non local goto: uses the setjmp() function to save the stack environment and longjmp() to jump to this saved environment. You can jump from one function to another using these instructions.
- uses nested functions: functions defined inside another function

Moreover, most of these cases are generally considered bad practice and should only be used in extreme case.

By default, gcc does not make any inlining unless you enabled the optimizations (*-O3*) or the specific inline substitution optimization (*-finline-functions*). You can also specify that a function has to be inlined even if the optimization is not enabled using the *always_inline* attribute.

Once the inline substitution optimization is enabled, GCC will not inline every candidate. It will heuristically choose which functions are simple enough to be inlined. Furthermore, GCC also has several limits (maximum size of the function that can be inlined, maximum compilation unit growth due to inline, maximum number of instructions for a function to be inlined, etc). All these limits can be modified using command-line. GCC may also choose to inline only some call sites to a function. It is possible that a call site is not eligible to inlining, for example, a recursive call within the definition.

Of course, the compiler has to know the body of the called function to inline it. That said, a member function has to be declared and defined in the body of the class in the header file to be inlined. And a global function also has to be defined in the header file. If a function is only declared in the header files, it may be inlined in the same compilation unit as where it has been defined.

If every call to a static function is inlined and the function address is never used, GCC does not output the assembler code for this function (unless you specified *-fkeep-inline-functions*). In the other cases, the function can be called from other source files, so that GCC has to output the assembler code for this function.

Some linkers can perform link-time optimizations, so they can inline some functions that the compiler does not consider because they have a whole view of the application. But most of the time, these linkers are still experimental. For example, GCC 4.6 has the option *-flto* to defers the optimizations to link-time. Nevertheless, some tests of the ATLAS software team have shown that this feature was not yet completely stable.

3 Information extraction

This chapter presents the implementation details of the information extraction part of the project. This part contains the graph reading, the call graph information extraction, the computation of information from the call graph, and finally, the extraction of information from the object files.

3.1 Graph manipulation in C++

The first operation to implement was to open a DOT file and convert it to a C++ object. Of course, it would have been possible to write everything from scratch, but graph theory is a complex subject and a lot of time can be saved and a lot of bugs avoided, using an existing library.

There are several operations that we need the library to do:

- open a DOT file
- add information to nodes and edges
- edit the graph (remove edges, nodes, edit information, ...)
- traverse the graph in different directions
- manipulate large graphs (the ATLAS call graphs have more than 40'000 vertices and 160'000 edges) in an efficient way

One more important requirement about this library, is that it must be an open source library and that the license includes no copyleft.

There is not a lot of known C++ graph manipulation libraries. Here are the ones we found:

- Boost Graph Library (BGL)[11]
- LEMON[12]
- igraph[13]
- OGDF[14]
- LEDA[15]
- AGraph[16]
- NGraph[17]

From these libraries, LEDA is not open source, AGraph and igraph are C libraries (they can be used from C++, but they do not use C++ capabilities), and NGraph is too limited to handle graph as big as ours, so that will allow us only three libraries in our case. We studied these libraries more in detail in order to to choose one of them. Table 1 summarizes the results of our comparison:

Table 1: Comparison of C++ Graph manipulation libraries

Criteria	BGL	LEMON	OGDF
Last update	March 2011	September 2010	October 2010
License	Boost Software License	Boost Software License	GPL V2 or GPL V3
DOT File Reader	Yes	No	No
Add information	Edges, Vertices	Edges, Vertices	No
Documentation	Very complete	Complete	Basic
Examples	A lot	A few	A few
Edit the graph	Yes		
Navigate the graph	Yes		
Performances	Very good	Better than BGL	No information
Other Features	More than necessary		
Ease of use	Complex	Complex	Relatively easy

This comparison is only valid in our special context, where we do not need a lot of specific features. For a project with specific needs on graph manipulations and algorithms, this comparison should have been done with deeper study.

For the performances comparison, we found a comparison including LEMON and BGL[18] showing that the LEMON library provides more efficiency, but BGL has excellent performances. We have not found any comparison or discussion about the performances of OGDF, but from what we saw, it should be less efficient than the two others in manipulating large graphs.

An important information we saw about these libraries is that they are overkill for what we need, but they are well-designed enough so we can use only a small part of the library without any problem.

After this little comparison, we chose to try the Boost Graph Library for several reasons:

- a DOT File Reader is already included in the library, so that a lot of time will be saved
- the library is complex, but we will use only a very little part of it, so that we avoid most of the complexity
- there are a lot of examples
- the documentation is pretty good, there is even a book about the library
- the library is well-known

- the library is maintained regularly
- it is part of the Boost C++ libraries which have a really good reputation in the C++ world
- the library uses modern C++ capabilities
- it is really fast

Our second choice would be to try LEMON, but BGL has the advantage of having a DOT graph file parser included. We have not chosen OGDF because it is not possible to add information to the nodes and vertices and there is not a lot of examples. Moreover, it also has drawing capabilities with graph layout that are not interesting in our project and that can make our work a bit harder.

The version of BGL used for this project is the 1.46.1. The project is compatible with Boost 1.32.0 and later. The older versions of Boost cannot be used because they do not provide bundled properties.

3.1.1 Boost Graph Library

3.1.1.1 Build BGL

The Boost Graph Library is a header only library, like most of the Boost C++ libraries. It means that they are only made of header files that you simply have to include in your application. You do not have to build it, just use it. There is only a little exception for Boost Graph, the DOT file reader that must be built.

So first of all, we have to include the header files of Boost into our application. Listing 13 shows the folder layout used for this project; it seems to be the most common layout for C++ projects.

Listing 13: Directory layout of the project

```
- root
 | - src (*.cpp sources files of the project)
 | - include (*.hpp header files of the project)
 |    | - boost (*.hpp header files of Boost)
 |    |     | - graph (*.hpp header files of Boost Graph Library)
```

Then, once we have the header files of Boost in our application, we can build the Boost library. This process is easily done using the tools provided by Boost. You just need several shell commands to launch the build and the installation of the library into your system, as seen in Listing 14.

Listing 14: Build Boost Graph Library

```
cd /path/to/boost_1_46_1/
./bootstrap.sh --with-libraries=graph
sudo ./bjam install
```

Boost Graph Library is now built and installed in your computer. In the next section, this installation will be tested.

3.1.1.2 Test BGL Installation

Once we have installed Boost, we can try if the installation was successful. For that, we used an example taken from the samples of Boost Graph to read a DOT graph file from a string stream:

Listing 15: Read a DOT file in memory

```
#include <boost/graph/graphviz.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <string>
#include <sstream>

using namespace boost;
using namespace std;

int main() {
  // Vertex properties
  typedef property<vertex_name_t, string, property<vertex_color_t, float> > vertex_p
      ;
  // Edge properties
  typedef property<edge_weight_t, double> edge_p;
  // Graph properties
  typedef property<graph_name_t, string> graph_p;
  // adjacency_list-based type
  typedef adjacency_list<vecS, vecS, directedS, vertex_p, edge_p, graph_p> graph_t;

  // Construct an empty graph and prepare the dynamic_property_maps.
  graph_t graph(0);
  dynamic_properties dp;

  dp.property("node_id", get(vertex_name, graph));
  dp.property("mass", get(vertex_color, graph));
  dp.property("weight", get(edge_weight, graph));

  // Use ref_property_map to turn a graph property into a property map
  ref_property_map<graph_t*, string> gname(get_property(graph,graph_name));
  dp.property("name",gname);

  stringstream gv("digraph {graph[name=\"graphname\"] a c e [mass = 6.66]}");

  bool status = read_graphviz(gv,graph,dp,"node_id");

  std::cout << "Status : " << status << std::endl;

  return 0;
}
```

And then, a simple makefile is used to build this little program:

Listing 16: Makefile to test BGL Installation

```
CFLAGS = -Wall -c -g
LFLAGS = -Wall -g

read_graphviz : read_graphviz.o
  g++ $(LFLAGS) -lboost_graph -o bin/read_graphviz bin/read_graphviz.o

read_graphviz.o : src/read_graphviz.cpp
  g++ $(CFLAGS) -Iinclude -o bin/read_graphviz.o src/read_graphviz.cpp

clean:
  rm -rf bin/*
```

This makefile compiles and links our program. We can run it directly:

Listing 17: Run the test

```
make
./bin/read_graphviz
```

And that should give us this result:

```
Status : 1
```

With a few commands, we have succeeded in compiling and running a program using Boost Graph Library, and more specifically, the part that interests us.

3.2 Parse the call graph

We saw in Section 3.1.1.2 how to parse a graph file in DOT language format. But this was an example using a graph with not exactly the same information as our graph from Callgrind. When using our tool chain to create the DOT file from the Callgrind profile data, there are some additional properties for a vertex, like *module*. If we try to read our DOT graph file using the last example, it will fail because Boost Graph Library has no way to parse and store these properties.

In order to parse our call graph in a valid graph object we must tell BGL how to parse the properties. Let's take the *module* property of a vertex as an example: First, we declare a new type property and install it for the vertices:

Listing 18: Declare a new BGL property

```
enum edge_font_color_t { edge_font_color };
BOOST_INSTALL_PROPERTY(edge, font_color);
```

We must follow the BGL naming conventions for our property, if we do not, the compilation will fail. Indeed, BGL uses macros, compile-time assertions and metaprogramming to verify the properties.

Then, we have to add this property to the list of properties supported for an edge. So,

Listing 19: Default vertex properties

```
typedef property < edge_weight_t, double > EdgeProperties;
```

becomes

Listing 20: Declare vertex properties

```
typedef property <edge_weight_t, double, property<edge_font_color_t, std::string>>
   EdgeProperties;
```

The properties are chained one after the other using templates. And last, but not least, we have to tell the reader how to bind the edge property to a property of the file, filling the dynamic properties:

Listing 21: Declare dynamic properties

```
property_map<Graph, edge_font_color_t>::type edgeFontColor = get(edge_font_color, *
   graph);
dp.property("fontcolor", edgeFontColor);
```

And we have to repeat this operation for every property of the edges and vertices. Once every property has been defined, we can parse the call graph the same way we did in Section 3.1.1.2. More than the properties of the call graph, we also have to repeat these operations for each additional information that we want to store for a specific edge or vertex.

3.2.1 Parse information

Once the graph has been parsed into a Graph object, we have to extract information from the labels. In the DOT file, the information is all contained into labels, as presented in Listing 22.

Listing 22: DOT file label formats

```
Vertex label: ${name}\n${inclusive_cost}%\n(${self_cost}%)\n${calls}x
Edge label: ${cost}%\n${calls}x
```

We need to extract is all the ${x} parts of the labels.

To simplify the temperature calculations, these sub parts of the complete information are parsed into other properties. To do that, we iterate on every edge and vertex to populate the specific properties parsing the label using basic string manipulation.

3.3 Compute information

Another information that we want to calculate is the frequency of each call site and function. We have the number of calls, but this is not relative to the current application. To have a value relative to our call graph, we have to compute frequencies. For that we need the total number of calls in the whole call graph. We cannot obtain this information directly, we have to iterate through every function or call site. There are at least as many call sites as the number of functions - 1, so we make the iteration on the functions to compute the total number of calls, to save some time. We first iterate through the functions to parse the labels and to compute the sum of the calls, then once again through the functions to compute the frequencies and then through the call sites to make both operations at once.

Computing the frequency of a function or call site is really easy, as shown by Figure 2.

$$
\begin{aligned}
total_{calls} &= \sum_{f \in callgraph} calls_f \\
frequency_{cs} &= \frac{calls_{cs}}{total_{calls}} \\
frequency_f &= \frac{calls_f}{total_{calls}} \\
f &: \text{function} \\
cs &: \text{call site}
\end{aligned}
$$

Figure 2: Computing the frequencies

3.4 Parse Object Files

The call graph lacks some features that may be useful for our analysis, like:

- the size of the functions
- the number of arguments of the functions
- the information if a function is virtual or not

This information is available directly on the object files.

We chose to test Dyninst[19] to extract this information. There are two ways to get information using Dyninst:

- parseAPI: extract control flow information from a binary file
- symtabAPI: extract information from a binary file using the symbol table

The advantage of the parseAPI is that it does not need a symbol table in the binary file.

Nevertheless, parseAPI has a big flaw. It does not recover the parameters of a function. If you have two functions with the same name (overload) in the same class, parseAPI will find two functions, but we have no way to distinguish the two functions. On the contrary, symtabAPI can get the names of the functions and even demangle the name to display a so-called "pretty name".

At first, we thought that this analysis would be possible even with a file without symbols, but it turned out that it will not be possible. Indeed, without symbols, even Callgrind has no way to add function names on the call graph. But, the good point is that ATLAS uses almost only shared libraries and the symbols we need in a shared library are considered global and are never stripped, so we can assume that the symbols will be present on the used libraries.

The size of the function is available from both libraries.

Because parseAPI does not have any information about the parameters of a function, there is of course no way to get the number of parameters. In the case of symtabAPI, the number of arguments is only accessible when the debugging symbols are available, but these symbols are stripped on a shared library because they are not global. We decided to compute this number by hand from the function name because neither of the two libraries provided a good way to compute the number of parameters. This computation is not difficult to perform when we have the debug symbols or the global symbols of a library. We have to count the number of commas in the function signature from the first left bracket to the next right bracket. We must take several special cases into account:

- a function with 0 parameters has the same number of commas than a function with one parameter, so we must verify that there is a parameter inside the function parameter list
- templates may also have commas, so we must not count any comma inside a template declaration
- a function may take another function in parameter so that we must not count the parameters of the second function

Given that neither of the libraries can give us information about the virtuality of a function and that symtabAPI give us more information about the function names, we chose to use this one for our analyzer. But after some time, we realized that there were some reasons to do the job ourselves:

- We parse only what we need to parse
- We parse the library or the executable only once
- We avoid the dependency on a heavy library. Indeed, parseAPI is depending on other Dyninst APIs that we have to install

This was not a big deal because, as we will see in the next two sections, the search for the virtuality and the search for the size of the functions are very similar. This is also more complexity added to the project.

3.4.1 Detect if a function is virtual

Neither of the two tools provide a way to know if a function is virtual from an assembly. After searching a long time for a library providing this feature, the only way we found to discover the virtuality of a function was to read the assembly using, for example, *readelf* or *objdump*, locate the v-table and find whether a given function is in the table indicating that it is virtual.

Of course, directly parsing the shared library file is by no way portable among compilers and architectures, so this is only applicable in our specific case, but the application can be extended so we parse files generated by another compiler or for another architecture. In this chapter, we are speaking of files generated by gcc 4.3.5 in a Linux system on an Intel architecture.

If we are working with an executable, the debugging symbols must be present to find this information, but with a library, all the needed information is global, so that we do not care about what is present or not, it is present in any case.

Both the executable files and the shared libraries are in Executable and Linkable Format (ELF). This format is relatively simple. It is made of one header followed by the file data organized in several segments and sections. The segments contain the runtime data and the sections contain the linkage and relocation data. It is a binary file and we need something to convert the ELF binary data into something we can parse or extract data from. Indeed, implementing the binary parsing ourselves would have taken too long.

The result of our search will be a list of functions that are virtual. In an ELF file, we can only find the mangled names of the functions and the symbols. We could also have demangled all the names, but the mangled name is lot shorter and so more interesting to store than the full name. This has a performance advantage because we will use the mangled names as keys, so that keeping keys shorter makes the search faster.

In a shared library, the addresses of the virtual tables are visible on the .dynsym section of the ELF file. A virtual table can be identified by its mangled name, starting with _ZTV, then the number of

characters of the class name and finally the name of the class the v-table refers to. The addresses of the functions are also in the same section and we can identify them using their mangled names. To read the content of the virtual function table, we have to look at the .rela.dyn section. The v-table starts with the virtual table address followed by the type information of the enclosing type. As a result, we have the information about the functions of the v-table. Every function is encoded using eight bytes in an ELF64 file and on four bytes in an ELF32 file. So we can easily identify whether function is present in the virtual table or not.

In an executable file, it is slightly more complicated. The list of the functions are all present in the .symtab segment of the file. Here again, we can identify the virtual tables from their mangled name. To read the content of the virtual table, we have to parse the .rodata segment. But in this case, it is the .rela.dyn section because the content of the virtual table is written four bytes by four bytes (or eight bytes by eight bytes for a 64 bits executable), on the reverse order byte by byte. So, we cannot directly bind the values from this segment to the values found in the symbol table, we have to parse the values to find the corresponding functions. Again, the virtual tables contain not only functions, but also type information for example.

The case of multiple inheritance is not difficult to manage. It means that there can be several virtual tables for one type. In the case of virtual inheritance, it also changes the organization of the virtual tables and their sizes, adding more offsets.

At the beginning of this research, we chose to use *readelf* directly from the C++ program to search the virtuality of a function. We chose not to use *libelf* directly because the library is complicated to use and in an old C-style. Our parsing is easy to do because we have at most two sections or segments to read per ELF file and the only operation we have to do is to compute information from some links and bind section together. It is not too difficult to make the string parsing directly by hand.

Some weeks after, we switched to an implementation using *libelf* for several reasons. First, we were depending a lot on the implementation of *readelf*. On some computers, the output of *readelf* is not the same, so our parsing would fail. Moreover, if the output of *readelf* changes from one version to another, we have to adapt our implementation to take into account the new version of the output and the analysis will not be backward compatible with the other versions of *readelf*. This is not an acceptable limitation of our program. Furthermore, forking to another program adds a non-negligible overhead. String computations have also a big cost that we can avoid working directly with the content of the ELF file. We also improved the algorithms in order to be more efficient. Section 7.4 shows the considerable performance savings using *libelf* in place of the old version.

The implementation using *libelf* is not easier, but the code more expressive, in that it expresses exactly what we are extracting from the ELF file. Another advantage of using *libelf* is that we can check that the format is really correct before parsing it, so we are sure to not fail during parsing. We are not dependent on a kind of output because the library exactly represents the content of the ELF file so that we are only dependent on the ELF format that generally does not change. Not using *readelf* means to not benefit for the advantages it offers. For example, all the symbols are automatically resolved to their string name in the *readelf* output. In our case, we had to read the symbols and bind them to their real names that we obtain from the string table section.

Detecting if a function is virtual is a good start, but it can lead to some false positives. Indeed, if we invoke a virtual function in a static way by specifying exactly the function we want to call, we will have no way of saying that the call is not virtual because we consider any call to a virtual function to be a virtual call. We can think of some ways to avoid this problem:

- We can read the instructions to identify how a specific call site is written and then detect if the call is virtual or not
- We can read the sources of the program to identify how the function is invoked

These two ways are expensive, For the first one, we need to parse the whole application assembly code and for the second we will have to read the whole source code. But even doing that, we will still be not sure of the result. We have to guess which statement is the call site because we cannot easily bind a call site to an exact statement in the function. We can imagine that in a specific function there are two call sites to the same virtual function, once called virtually and once statically. We can also face the problem of inlining. Indeed, in the source code, it is possible that there are several call sites to a function, but in the assembly there are only remaining call site to the function.

Even if neither of these ways is perfect, using one of these solutions will improve the detection, but will also slow down the analyzer because we will have to parse the source source files or the binary instructions to find information for every call to a virtual functions. In our case, we decided to accept these false positives because they should be rare and the cost of avoiding them is higher than the benefit of detecting them.

3.4.2 Find the size of the function

As we were going to parse the ELF ourselves we had to find a way to find the size of the functions reading the ELF file. It turned out, this is quite simple to do. In an executable or a shared library, the function list is kept in the symbol table. Every line of the symbol table represents a symbol, so we have to parse the whole symbol table line by line to find every function. Identifying a symbol as a function is really straightforward: it is identified by the type FUNC. The symbol contains also the demangled name of the function and the size, decimal based.

The only difference for this search between a shared library and an executable is the name of the symbol table: .symtab for an executable, .dynsym for a library.

4 Information analysis

This section describes all that was done in order to analyze the information that has been computed from the call graph and the object files.

4.1 Temperature

To choose which call sites and functions to inline, we apply a heuristic to all call sites and functions of the program. The interest of inlining a specific function or call site is called its temperature.

The temperature function is not exactly the same for a call site or for a function because we do not have the same information. Indeed, when we are looking at a function we have no information about the library overhead. Also, it is generally more interesting to inline a specific set of call sites to a function than the whole function.

Figure 3 shows the computation of the temperature of a given function.

$$
\begin{aligned}
temperature_f &= \frac{cost_f}{size\_overhead_f} * frequency_f \\
cost_f &= 1.0 + parameters_f * 0.1 + virtuality_f \\
virtuality_f &= \begin{cases} 0.39 & \text{if } f \text{ is virtual} \\ 0.0 & \text{otherwise} \end{cases} \\
parameters_f &= \text{number of parameters of } f \\
size\_overhead_f &= 1.0 + \frac{size_f * (in\_degree_f - 1)}{total_{size}} \\
size_f &= \text{size of } f \text{ in number of bytes} \\
total_{size} &= \sum_{f \in callgraph} size_f \\
f &: \text{function}
\end{aligned}
$$

Figure 3: Function temperature

First of all, to calculate the temperature of a function, we have to calculate the cost of its invocation. To do that, we take 1.0 as the base cost of invoking a function and then we consider other parameters increasing the cost of the call. That is calculated by $cost_f$. The unit of $cost_f$ is a factor representing the cost of the call compared to the cost of invoking the most simple function. To see where do the numbers in $cost_f$ come from, see Section 7.8.

The function temperature represents the interest of inlining based on the benefits (the avoidance of the invocation cost) and the overhead of inlining it. The overhead of inlining a function is calculated as a ratio of the size growth of the whole application. We have to multiply the size of the function by the number of call sites that call this function. We can remove once its size because if we inline all the call sites of a function, they do not have to keep the function and we save the size of the function itself. There are some cases where the function will not be removed, for example, if the function pointer is used or if the user specified some command line option, but we do not consider these cases. This overhead is represented by $size\_overhead_f$ and once again, the unit of this value is a factor representing the increase of the application size if the function is inlined. And finally, we consider the most frequent functions as more interesting using the frequency of the function.

Figure 4 shows how the temperature of a given call site is computed.

$$
\begin{aligned}
temperature_{cs} &= \frac{cost_{cs}}{size\_overhead_{cs}} * frequency_{cs} \\
cost_{cs} &= 1.0 + parameters_{dest} * 0.1 + virtuality_{dest} + library_{cs} \\
library_{cs} &= \begin{cases} 0.39 & \text{if } library(src) \neq library(dest) \\ 0.0 & \text{otherwise} \end{cases} \\
parameters_f &= \text{number of parameters of } f \\
size\_overhead_{cs} &= \begin{cases} 1.0 & \text{if } in\_degree_f = 1 \\ 1.0 + \frac{size_{dest}}{total\_size} & \text{otherwise} \end{cases} \\
total\_size &= \sum_{f \in callgraph} size_f \\
f &: \text{function} \\
src &: \text{the caller function} \\
dest &: \text{the callee function} \\
cs &: \text{call site}
\end{aligned}
$$

Figure 4: Call site temperature

As with the function temperature, we start by calculating the cost of the function call (the cost we will avoid by inlining the call site). The calculation of $cost_{cs}$ is almost the same as $cost_f$ with the only difference being that we have one more piece of information here. We know if the call is between two libraries or not. If this is the case, we add a new overhead to the $cost_{cs}$. So the unit of $cost_{cs}$ is also a factor representing the overhead of this call compared to a simple function call. The numbers present in this equation are explained in details in Section 7.8.

Next, we have to calculate the size overhead of the inlining. This overhead is simple to compute because

we know that only one call site has to be inlined. If this call site is the only call site referring to the function, there is no overhead because the function will be removed. The $size\_overhead_{cs}$ is also a factor indicating the increase of the size of the whole application.

And finally, we are using the frequency of the call site ($frequency_{cs}$) to ponder the temperature so that a frequent call site will be more interesting to inline than a rare one.

4.2 Library issues

One of the goals of the project was to discover the functions that should be in the same library. To find these functions, we chose a simple algorithm. Figure 5 shows the pseudo code of this algorithm.

```
1: callSites ← every call site from src to dest with library(src) ≠ library(dest)
2: for all cs in callSites do
3:   if calls(cs) ≥ Max then
4:     src and dest should in the same library
5:   else if calls(cs) ≥ M then
6:     find every path from dest to library(src) using edges in library(dest) with length(path) ≤ L
7:     for all path in paths do
8:       if minCalls(path) ≥ P then
9:         src and dest should in the same library because of path
10:      end if
11:    end for
12:  end if
13: end for
```

Figure 5: Find library issues algorithm

There are some lines that deserve more explanations. On line one, we iterate through every call site and we only get those where the caller lies in a library different from the callee. For line six, we use a depth-first recursive search with a maximum depth of L to compute all the paths between *dest* and a function in the library of *src*, restricting the vertices to the functions only in the library of *dest* and *src*. Finally, at line eight, we consider the calls of a path as the minimum number of calls for each call site.

Several parameters are used to limit the number of results of this algorithm:

- Max: The maximum number of calls allowed for a call site between two libraries
- M: The minimum number of calls to consider a call site between two libraries as a candidate
- L: The maximum length of the path between *src* and *dest*. We use this parameter to improve performances of the algorithm. Indeed, if L is a big value, we will follow some very long paths and test every sub path of it.

- P: The minimum number of calls to consider a path between *src* and *dest* as an issue

We use the parameters M and P to concentrate our efforts on the more important issues. In an application where the hot functions are called millions of times, it is generally not interesting to be warned about an issue with a path called three times.

As you can see, this algorithm detects two different things:

- Very frequent call sites between two libraries. It can be an issue if a specific call site called millions, or billions, of times between two libraries. If the source is not called often or the target do not call a lot of functions, it can be very interesting to move the source, respectively the target function in the other library.
- Frequent call sites between two libraries that are part of circular link between these two libraries

After we found a library issue, it can be solved by moving a function from one library to another. But we have two choices: we can move the function from A into B or the function from B into A. We have enough information to decide programmatically which function should be moved to which library. Indeed, moving a function F from X to Y has two consequences:

1. Every call to F from a function in X and every call from F to a function in X will now be library calls
2. Every call to F from a function in Y and every call from F to a function in Y will now be a normal calls

The first point adds more overhead to the application and the second removes overhead. So, the interest of moving a function from one library to the other is only a matter of counting avoided and new calls. Figure 6 shows in details the computation of the interest for a specific function moving from X to Y.

$$
\begin{aligned}
interest_f &= benefit_f - cost_f \\
benefit_f &= |\text{calls to f from Y}| + |\text{calls from f to Y}| \\
cost_f &= |\text{calls to f from X}| + |\text{calls from f to X}|
\end{aligned}
$$

Figure 6: Library candidates

Now to choose which function to move, we have to compute the $interest_f$ for both choices and choose the one with the biggest. If both interests are less than or equals to zero, none of the choices are good. If there is a path with this library issue, we can also consider parts of the path as candidates to be moved. It is perhaps interesting to move members of the paths and not only *src* and *dest*. What we will test will depend on the case we choose, between:

1. Moving *src* to the library of *dest*
2. Moving *dest* to the library of *src*

In the first case, we should also consider moving the end of the path, the other function in the same library as *src*, to the library of *dest*. Of course, we have to test if the interest of moving it is greater than zero. In the second case, we should consider the middle of the path (everything from *dest* to the last call site, excluding these two) as candidates to be moved to the library of *src*. Once again, we have to compute the interest of moving these functions. We decided not to break the path, namely if a function in the path is not interesting to move, then we do not consider the following functions as candidates to be moved in order to avoid breaking the path into multiple libraries.

Having a path going from a library A to a library B and then going again to A means that there is a circular dependency between A and B, which is often a sign of a bad design. So we chose to profit from the algorithm to detect every circular dependency between libraries. To detect every library circular dependency, we can use the same algorithm giving the parameter specific values:

- M = 1
- L = inf
- P = 1

Doing this, we will find all the circular references between two libraries. The problem with this algorithm is that it can take a long time when analyzing a large call graph. What we really want is to find the cycles among the libraries. If we represent the dependencies and the libraries in a graph, we have to find the graph cycles and we have found the circular dependencies. The algorithm is detailed in Figure 7.

```
1: create a new graph G representing the libraries
2: for all callSite in callGraph do
3:     src ← library(source(callSite))
4:     dest ← library(target(callSite))
5:     if G does not contain src then
6:         add new vertex (src) in G
7:     end if
8:     if G does not contain dest then
9:         add new vertex (dest) in G
10:    end if
11:    if src ≠ dest then
12:        add edge between src and dest in G
13:    end if
14: end for
15: circularDependencies ← find every cycle in G
```

Figure 7: Circular dependencies algorithm

In line 15, to find the cycles in the graph, we use the Tarjan's strongly connected components algorithm[20]. This algorithm has a time complexity of $O(|vertices|+|edges|)$, so that the time complexity of our whole algorithm is $O(|callsites|+|libraries|+|dependencies|)$. It is a lot more efficient than the last algorithm with specific values of M, L and P.

4.3 Clustering

An idea that was mentioned at the very beginning of the project was finding clusters of hot functions. The idea was that inlining specific clusters at hot points of the application execution could be more interesting than inlining single call sites across the whole application. A call site is considered hot if its temperature is high. In our case, a cluster is represented by a vector of call sites.

The algorithm is relatively simple, as shown in Figure 8.

```
1: copy ← copy of the call graph
2: for all callSite in copy do
3:    if temperature(callSite) < M then
4:       remove callSite from copy
5:    end if
6: end for
7: clusters ← find every group of connected call sites in copy
```

Figure 8: Clustering algorithm

Basically, the graph is copied. Then, all the call sites that are not hot are removed from this copy. This graph should now normally be constituted of several groups of call sites. Then, we have to find these groups on the graph. To find these clusters, we use the algorithm described in Figure 9.

```
1: clusters ← empty list of clusters
2: for all function in graph do
3:    if !visited(function) then
4:       cluster ← empty cluster
5:       findCluster(function, cluster)
6:       if size(cluster) > 1 then
7:          if size(cluster) > S then
8:             reduce size of cluster to S
9:          end if
10:         add cluster to clusters
11:      end if
12:   end if
13: end for
14: return clusters
15:
16: findCluster(function, cluster)
17: if !visited(function) then
18:    for all cs in out_edges(function) do
19:       add cs to cluster
20:       findCluster(target(cs), cluster)
21:    end for
22:    for all cs in in_edges(function) do
23:       add cs to cluster
24:       findCluster(source(cs), cluster)
25:    end for
26:    mark function as visited
27: end if
```

Figure 9: Find clusters in a graph

There are two parameters used to limit the results:

- M: The minimum temperature value after what a call site can be considered hot
- S: The maximum size of a cluster to avoid having too big clusters

Once we have found a cluster, we want to inline every call site contained in the cluster. The bigger the cluster the harder it is to inline it. Therefore, we have to reduce the size of the cluster. For that we chose to add a maximum size to the clusters. When a cluster contains more than S elements, we reduce it by keeping only the hottest call sites. We also chose to not consider a cluster containing only one call site because this call site will also be found in the list of the hottest call sites.

Another thing a user will perhaps do with a cluster is review this specific part of its code before doing

anything else. A cluster is a hot group of call sites, so improving this application part can reduce the temperature of the whole application.

4.4 Virtual hierarchy issue

Given that we have all the information about the virtual functions of the application, we can verify the virtual functions hierarchies to see if they are really justified. What we call a virtual function hierarchy is a set of all functions overriding a virtual base function (including this base function). For example, the class B overrides *z()* of the class A, so we have two versions of the same functions, creating a virtual function hierarchy.

The first operation we have to perform is to separate each virtual function hierarchy. The technique used can produce several false positives because we do not have exact information about the inheritance. The technique is simple. For every function name (without the type indication), we store every function. Figure 10 shows the pseudo code of this search.

1: **for all** $function$ in all virtual functions **do**
2: $pos \leftarrow$ start position of the function name
3: $functionName \leftarrow substr(function, pos)$
4: insert $function$ in the $hierarchy$ of $functionName$
5: **end for**

Figure 10: Search the hierarchies of virtual functions

Once we have all the hierarchies, we have to calculate the number of times each function has been called. We have to fill a map of function calls from the call graph because not every function is in the call graph, and then we can match the number of calls for every virtual function of the hierarchy.

We do not consider the groups with no called functions because they have no cost and are not interesting to optimize.

We can now search for some patterns that can be issues pertaining to virtual function hierarchies. Here are the patterns we look for:

- Only one function is called on the hierarchy
- One function is called more than X% of the time
- Less than Y% of the functions are called
- The hierarchy contains only one function

In these cases, we may consider the possibility that there is something wrong in the virtual hierarchy.

Because of the lack of information about the inheritance tree, some hierarchies are almost always present, like the destructors and the default constructors. We can easily filter these hierarchies using the demangled name because they are not of particular interest at this point.

4.5 Statistic reports

Because we have a lot of information about functions available in the application, we thought that it was a good idea to produce some *statistic reports*. A statistic report is a list of functions or issues that we give to the user. This can be just some information, but this can also point to some bad practices.

Of course, the first reports to be displayed are the ones showing the hottest call sites and functions of the applications. But we display some more reports not depending on the analysis. For example, here are some statistic reports provided by the application:

- The functions that take most of the run time
- The biggest functions in the application
- The tiniest functions in the application
- The functions with the greatest number of parameters
- The most called function and call sites

These statistic reports can be helpful to the user in order to have a better knowledge of the analyzed application.

When applied to program performances, the 80-20 rule (also known as the Pareto principle) states that, in general, 80% of time spent is only on 20% of the code. So, we added a statistic report to the application displaying in what percent of the functions 80% of the time is spent and displaying the list of these functions. It turned out that this percentage was way lower than 20% in our analyzed applications.

More than simply giving lists of information, we also reported an issue that can be easily detected: functions with too many parameters. By default, a function is considered over-parameterized if it has more than 10 parameters, but this value is configurable. With all the available information, we can add new issues in the future.

5 Design

During this project, we chose to follow an object-oriented style for the code. We represented the part of the call graph using objects (Function, Callsite) and created several classes to manage the different part of the analysis. We also tried to use a design that will be easy to use by the developers that want to use the analyzer as a library.

Figure 11 shows an UML class diagram of the library.

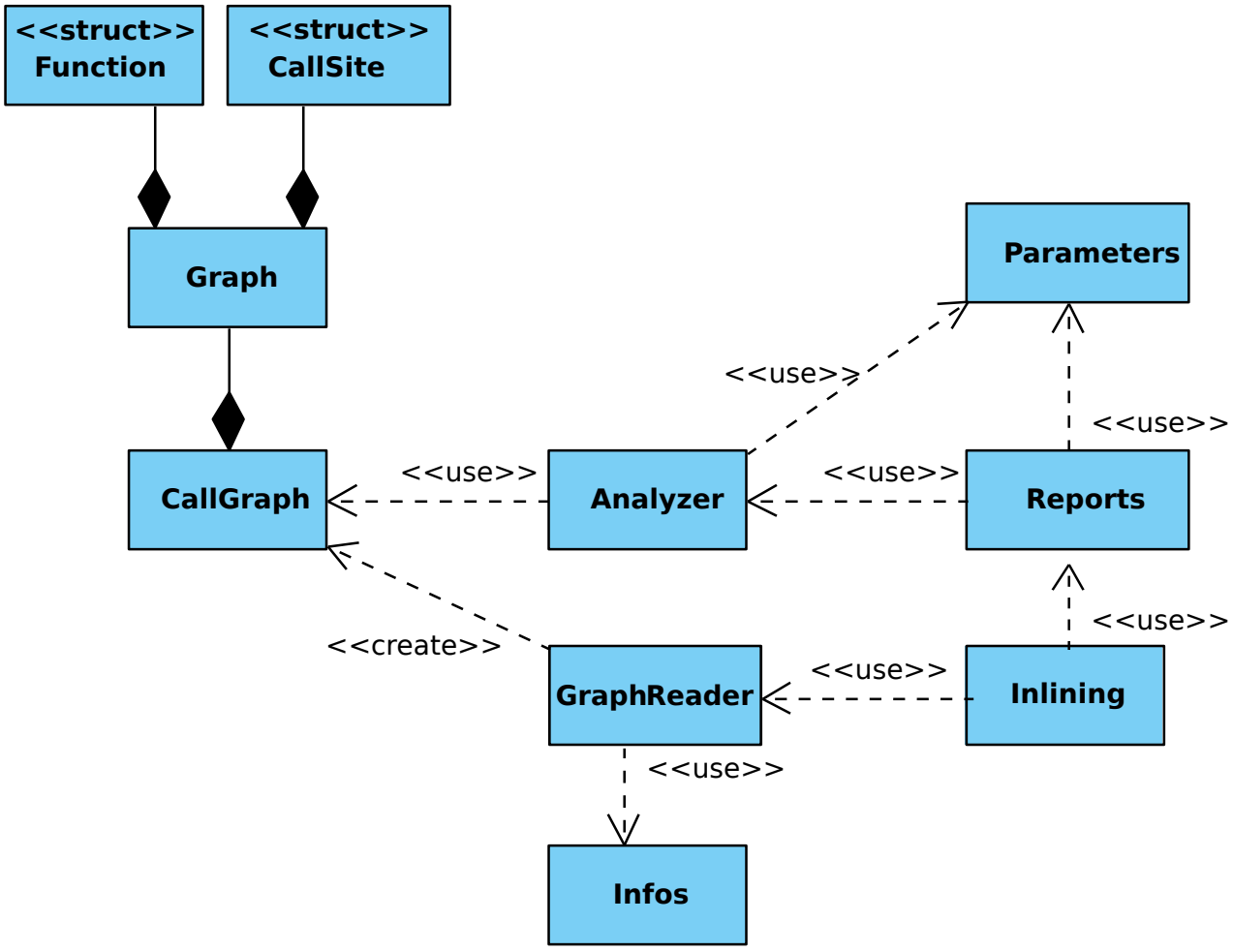

Figure 11: Class diagram

Here are some explanations about the classes and their roles in the application:

- *Graph*: The basic Boost Graph Library graph structure.
  - *Function*: Represents a function in the graph, it is used as identifier for a function.
  - *CallSite*: Represents a call site in the graph, it is to identify a call site.
- *CallGraph*: Class used to encapsulate the raw graph and provide some utility functions to make easier the manipulation of the graph. This is also the class that we use to get information about the call sites and functions.
- *Inlining*: Helper class to launch a full analysis on a given call graph file.
- *GraphReader*: This class is responsible to read the graph from the DOT file and then to parse the information of the labels.
- *Infos*: Class used to parse the ELF files. Contains the information about the virtuality and the sizes of the function. We can also use this class to get the list of all the virtual functions of the application.
- *Analyzer*: Contains all the functions necessary to analyze the call graph. This is the class responsible for all the algorithms like the temperature calculations, the clustering or the library issues. This class contains only functions that return data, no reporting functions.
- *Reports*: Display the information extracted by the analyzer and output some statistic reports produced by reading the call graph. This class only outputs to the console what it can get from the *Analyzer* or from the *CallGraph* fro the statistic reports.
- *Parameters*: Contains all the configurable parameters of the algorithms. Can be used to set the values of the parameters or to retrieve them. This class is used by the *Analyzer* and the *Reports* classes in order to compute or display information.

With this design, the user is able to use the analyzer in two ways:

1. basic: Use the *Inlining* class, configure the parameters and launch the analysis. The result will be the same as invoking the tool from the command line, but everything is configurable programmatically. This can be useful to integrate the analyzer inside another command line tool.
2. complete: Use the other classes and choose exactly what the application wants to do on the call graph and which analysis have to be launched. This is useful if the user wants to customize exactly what should be printed or get the information and analyze it.

6 Tests

This section describes the tests that have been done in order to see if the results provided by the analyzer were interesting or not.

6.1 Tests on a real life application

Once the analyzer has become mature, it was time to start testing it.

Generally, the inlining tools are working at the compiler level. It seems that it is the compiler that chooses directly what to inline, generally heuristically. Testing them is easy. To test a change in a heuristic, we have to compile the analyzed application again and compare the execution time of the new application with the old compiled version. In the case of this application, it is a bit more complicated. Indeed, we are not working at a compiler level, the issues are intended for the developer and it is up to him to implement the issues in the analyzed application.

So, if we want to compare two versions of a temperature function, we have to analyze the profiled application twice (with both versions of the analyzer) and to apply the proposed solutions for both versions. And then compile both versions and compare the execution time of the two resulting executable files. It appears similar to the way a compiler level heuristic is tested, but the difference is that for us, the issues have to be resolved directly by hand in the source code. This implementation requires a good understanding of the analyzed tool and the used compiler and, of course, requires a lot more time.

Generally, it is better to test an analyzer like this one on several different applications, but we did not have enough time to test the analyzer on several tools, so we decided o choose only one tool. To choose the application to test the analyzer on, we decided that the tool:

- had to be open source
- had to be enough complex to be interesting
- had to be object-oriented
- had to be easily testable
- had to be easily modifiable
- had to run enough long to provide good benchmark results
- ideally had to work on different inputs

Based on these criteria, we decided to use CLang[21] as the application to test our analyzer on. CLang is a compiler for C, Objective-C and C++ based on LLVM[22]. Both LLVM and CLang are heavily object-oriented and the source code is very well maintained and commented. We can take advantage of

the fact that it is easy to test it on several different inputs to see the impact of our optimizations on input different from the one used when we profiled the application.

The tested version of CLang is the revision 134999 checkout from the LLVM SVN repository.

Once the tool has been chosen, we used Callgrind to generate a call graph of the compilation. We compiled the analyzer itself with CLang. Then, we analyzed the call graph to find the issues and to test implementing them to see the impact of performances.

The detailed results of the analysis are available on the Section 12.2.

We decided to create several versions of CLang to bench them and test the saving of the inlining. Here are the versions of CLang we tested:

- *base*: The normal version of CLang
- *cluster_1*: Inline the first cluster
- *cluster_2*: Inline the second cluster
- *callsites_1*: Inline the hottest call site
- *callsites_2*: Inline the 2 hottest call sites
- *callsites_3*: Inline the 3 hottest call sites
- *callsites_4*: Inline the 4 hottest call sites
- *callsites_5*: Inline the 5 hottest call sites
- *functions_1*: Inline the hottest function
- *functions_2*: Inline the 2 hottest functions
- *functions_3*: Inline the 3 hottest functions

The benchmark is made using a simple shell script. The script launches 100 times each version of the compiler and then computes the mean of the execution time. For each compilation unit, the compiler is launched twice at the start without timing it in order to be sure that the files are cached and to not discriminate the other runs. The versions of CLang are tested on the compilation of two programs:

- The analyzer itself
- The EDDI compiler[23]

The results of the benchmark are presented in Table 2.

Table 2: CLang performances

Version	Analyzer		eddic	
	Time [s]	Difference	Time [s]	Difference
base	5.11713	0	0.98423	0
cluster_1	5.08705	-0.588 %	0.98256	-0.170 %
cluster_2	5.1034	-0.268 %	0.97007	-1.439 %
callsites_1	5.11585	-0.025 %	0.98123	-0.305 %
callsites_2	5.1129	-0.083 %	0.98093	-0.335 %
callsites_3	5.11245	-0.091 %	0.97708	-0.726 %
callsites_4	5.09759	-0.382 %	0.97614	-0.822 %
callsites_5	5.08783	-0.573 %	0.97574	-0.863 %
functions_1	5.09727	-0.388 %	0.99382	+0.974 %
functions_2	5.03564	-1.592 %	0.97889	-0.543 %
functions_3	5.027	-1.761 %	0.97589	-0.847 %

In the case of the compilation of the analyzer itself, every change led to some performance savings. The clusters seem interesting. Indeed, the first cluster has a temperature corresponding to the first four call sites and the result of inlining the first cluster is better than inlining the first four call sites. The second cluster also produces good results and its temperature is twice as low as the first one. But the most powerful optimization is to inline whole functions. We have more than one percent and a half performance improvements by inlining three functions. Of course, inlining a whole function has generally a greater effect on the size of the executable files.

The savings of inlining a call site or a function is not totally relative to the temperature. The first call site is not the one that allows the biggest savings and the situation is the same for the function. This can be explained by the fact that our heuristic do not take into account all the parameters that influence the cost of the function call. It is also possible that the heuristic can be improved to weight differently some of the parameters.

Even if the results are less important in the case of compiling *eddic*, there are some savings. The only case that does not result in some savings is the inlining of the first function that results in a loss of about one percent. We can also see that, the second cluster is several times better than the first one. For the call sites, it is even more interesting to inline them in the case of *eddic*, but inlining functions does not produce as many results as on the analyzer.

In our case, the executable growth is not really important, as shown in Table 3.

Table 3: CLang executable size

Version	Executable size [bytes]	Difference
base	41612841	0
cluster_1	41625171	+0.0296302769%
cluster_2	41630316	+0.0419942488%
callsites_1	41624451	+0.0279000417%
callsites_2	41624427	+0.0278423672%
callsites_3	41625051	+0.0293419043%
callsites_4	41625051	+0.0293419043%
callsites_5	41624463	+0.027928879%
functions_1	41606656	-0.0148632005%
functions_2	41631478	+0.0447866561%
functions_3	41631874	+0.0457382854%

The code growth is not directly related to the number of call sites inlined. Indeed, inlining three functions should grow the code size much more than inlining eight call sites of the second cluster, but this is not the case. This is due to the optimizations the compiler can do when we inline something. Moreover, the instructions needed to make the function call are removed from the code, so very small functions are smaller than the instructions necessary to invoke them.

In this specific case, the order of magnitude is really small between both versions, but in some cases, it is possible that it turns out to be an issue, is especially in embedded software where the code size is critical.

To conclude these tests, the solutions given by the analyzer are producing good results in the case of CLang. We can also say that the results are strongly dependent on the input with which we profiled the applications. It is very important to optimize the applications with the most general input. Furthermore, here we tested only one application and it is not proved that the analyzer will produce the same results for another application, the results can be worse or even better.

These tests show that the clusters are interesting without being spectacular. Indeed, at the same temperature, it is better to inline a cluster than single call sites. The problem is that, for the same temperature, we had to inline more call sites for the clusters than for the call sites. It is also possible that is some applications the clusters are just containing the hottest call sites.

Somebody may think that the results are not really important, but by inlining only five call sites we were able to save half a percent and inlining three functions saved more than one and a half percent. The saved time looks ridiculous here, but when we compile large programs the run time can be important, so even little savings can save a lot of time when we are compiling often. If we take the ATLAS software as an example, some jobs are taking hours to process. When every hour of computing has to be paid, one percent starts to be an important number. 1% saving on the ATLAS software allow the experiment

to save 100K$ per year. As the project was targeting large-scale programs, we can assume that the run time of such applications is generally quite big and expensive.

6.2 Tests on ATLAS

After some time, I had the opportunity to test the analyzer on ATLAS itself. This was a good test for the analyzer because the application has been designed for this kind of application which is large scale object-oriented programs. Moreover, ATLAS is very large, so it is also a good performance test to run the analyzer on an ATLAS call graph.

The whole analysis was made on a Linux 64 bits machine located at CERN in Switzerland.

Once we solved some problems on the machine, and on the code, we managed to run the full analysis on the ATLAS call graph.

The provided call graph contains 44'431 functions and 164'918 call sites. The number of libraries used in this call graph is 790. This is not the kind of graph we had tested before, so as explained in Section 7.7, it was an excellent opportunity to test the performances of the analyzer.

We have not implemented the results in the ATLAS source code. It would have taken too much time to implement the solutions and then to compare different versions. Not only changing the source code is hard, but compiling the whole ATLAS software is a long process and running it is even more time-consuming, so benchmarking several versions of ATLAS was not conceivable.

So, instead of implementing the results, we submitted them to the ATLAS software development team that studied them. They know the software very well and have already run several profiling tools and analyzer on it, so it was easy for them to evaluate the relevance of the results. Finally, it turned out that the results were very close to what they knew were critical in the ATLAS software. Also, they were very interested by the results that they had not before, like the virtual hierarchy issues or some library issues.

6.3 Generated application

As we saw that the tests of the analyzer would be very hard, Noé Lutz, the project's expert, had the idea of generating an application from the call graph and then executing the tests on the generated version instead of the real application.

This approach has some advantages:

- it is easier to edit the generated application than to edit the real one that we do not know well.
- it is faster to compile
- it is easy to bench (some programs are not easy to bench, especially if they create multiple processes or threads, if they take a special kind of output, if they take very long to run, etc.)

Of course, the savings we can obtain in the generated application are absolutely not guaranteed to be the same in the real one. It is possible that improving the performances of the analyzed version can lead to a loss of performance in the real one.

As the idea was interesting and quite original, we decided to spend some time to study the idea of an application generator. The idea was to generate a program close to the original based on the call graph and shared objects information. The point is that the call graph of the two applications should be the closest possible.

It turned out that it was far from easy to generate an application close to the original one.

If we want to define a function, we have to decode the function signature to know what the parameters' types are. Indeed, we cannot only put the demangled name as the name of the function because we have to know the exact type of the parameter. If it is not a built-in type, it will not exist in our application. A solution is to parse the object files in order to find the type information and to reconstruct classes and structures, but there is no guarantee that the compiler will output the type information and it is very compiler specific and it will be hard to reconstruct them using only assembly information. Even the use of the standard library is problematic because we have to know which package we have to include in the file to compile it. Moreover, the mangled name of a function does not include any information about its return type, we have no other choice but to choose arbitrarily one return type for each function. The easiest way to reconstruct the classes and the return types of the function would be to parse the source code of the application, but that kind of parsing is expensive because there are a lot of files and C++ is one of the hardest languages to parse.

In the case of a member function we also have to reconstruct the object and put the function as a member of this object. But if we do that, we have to manage instances of the object to invoke the function. So, we have to analyze the call graph to find the constructor calls to create new objects and then use them at the right place to invoke the functions. But the information can be misleading, because there are a lot of implicit copies and destructions that are made. If the function is virtual, we also have to recreate a virtual hierarchy of functions and an inheritance tree with possible multiple or virtual inheritance.

Once we have all the generated function declarations, we have to define them. That involves other issues. We do not know the order in which the call sites are invoked. Also, we do not know if the function invokes each of its call sites each time it is invoked. For example, a function can call the first call site 200 times the first time the function is invoked and then only call the second call site the next times it is invoked. It is very hard to calculate the number of times we have to call each called functions from a function. We can only make approximations and these can lead to a loss of precision because we will add some conditional statements or loop to emulate the behavior of the application.

Once we succeeded in recreating the structure and the calls among functions, we have to make these functions doing something. Indeed, if we put only function calls in the function body, we will not have something near the real application. We can generate some basic code in each function in order to match more or less its sizes. But of course, the goal is not to recreate the code, we only want that the cost and the size are relative to the original one. It is sure that the time passed in the functions will not be the same as the real one, but that is not a disadvantage because a faster application will be more convenient to profile.

Another problem would be if the profiled application is multi-threaded. If we use the call graph from Callgrind, we will have no information about the threads, so we can only generate a single threaded application that can be very different from the base version. There are no good solutions to this problem, it is a limitation that we have to accept on the generated version.

If we want to generate the closest possible application, we have to create several libraries, compile them and configure the linker. The best solution will be to generate a makefile to build each library and the executable automatically.

Of course, we want to prevent the compiler from making too many optimizations that could break our call graph. The best solution is simply to disable all the optimization flags of the compiler.

To conclude, the idea remains interesting and it could be a good track to test automatically some changes in the analyzer, but writing a good application generator would take a long time in order to have a generated version enough similar to the original. That could be an entire project.

7 Performance Analysis

The call graph of a standard ATLAS reconstruction job is a large graph with about 40'000 vertices and 160'000 edges. So, we have to pay attention to the fact that the analysis of this graph has to be made in a reasonable amount of time.

Generally, the graphs are put in two main categories:

- Sparse: when the number of edges is about ten times the number of vertices
- Dense: when the number of edges is about the square of the number of vertices

In the context of call graph, the graphs are generally very sparse. So for this benchmark we will take only sparse graph in consideration. For our benchmark, we will test the performances for graphs of size 100, 1000, 10000, 40000, 100000. The graph of size 40000 is here because it is about the size of an ATLAS benchmark. At the beginning, we also wanted to test a graph with one million edges, but it turned out that the computer used for the benchmark did not have enough memory to analyze a graph of this size.

All the results presented in this section are based on a computer with an Intel Core i7 at 1.73Ghz on an Ubuntu 11.04 and the sources are compiled using gcc 4.3.5 in 64 bits.

The graph used for this benchmark is an *adjacency_list* using *vecS* for storing both the edges and the vertices. The graph uses bundled properties to store the information of every edge and vertex.

The benchmark is really simple. Every tested part is executed on five different graphs and the mean of the different executions is computed. We take five different graphs because the efficiency of some algorithms depends on the input so that the results are more accurate if we take several input files.

The tested graphs are randomly generated by a little tool written for this purpose. It creates a DOT file with the good number of edges and vertices. The functions have a random number of parameters between 0 and 10. The function names is the same except for a numeric index appended to the function name. The number of calls of the call sites and functions is also randomly generated. To add an edge, we take two random vertices and create a new edge between them. The costs are randomly generated with a max value based on the application size. Finally, we generate a number of libraries depending on the number of functions and we randomly bind them to the functions.

When the benchmark is running, the libraries do not exist, so the function sizes and the virtuality are randomly generated. The function size is a random number from 1 to 1000 and a function has one on six chance to be virtual.

7.1 Importing the whole graph

First of all, the import of the whole graph from the DOT file involves creating a Graph, reading and parsing a file and adding many edges and vertices to the graph.

Table 4 presents the result for importing a whole graph with the Boost Graph Library.

Table 4: Importing a DOT graph performances

Number of vertices	Time [s]
100	0.048
1000	0.455
10000	4.031
40000	16.855
100000	43.666

It is a costly operation, involving a lot of file file reading and memory allocations. However, the time is increasing linearly so we are sure that the needed time will not explode for some graph sizes.

In most cases, this operation the most important part of the runtime. In the case where there are many shared libraries to parse, the parsing consumes more time than the reading operation.

7.2 Parsing information about the graph

Here, we calculate the time needed to parse the information contained in the graph. The computations are done with the information contained in the call graph, like frequency, labels parsing, etc. In the real analyzer, these operations are done in the same time as the graph importing. Here, we decoupled both parts in order to bench them separately. Table 5 summarizes the results of this benchmark.

Table 5: Performances of parsing the graph

Number of vertices	Time [ms]
100	6.955
1000	26.4922
10000	266.178
40000	1041.000
100000	2608.780

It is a very fast operation. It is a matter of iterating through every edge and vertex, parsing some string information and computing some totals. The time complexity of the parsing is linear in $O(|V| + |E|)$.

7.3 Navigating through the whole graph

We tested some graph traversal algorithms to see how long these computations take. It is not an operation of the analyzer, but it is an important operation used by almost every part of the application.

First, we tested a simple *foreach* loop on every vertex and then on every edge using the iterators of the graph. For each element, we get the value of a bundled property. Table 6 shows the benchmark results.

Table 6: Performances of *foreach* loops on graph

Number of vertices	For each vertices [ms]	For each edges [ms]
100	0.0072	0.0812
1000	0.0544	0.6354
10000	0.5572	6.8290
40000	2.1848	27.0398
100000	5.4716	68.2002

The order of magnitude between the two traversal techniques is a bit more than ten. This is probably because the edge descriptor is three times bigger than the vertex descriptor. The edge descriptor conveys information about the source and the destination of the call site and the identifier of the edge. But, even for a very large graph, these operations are fast.

Then we tested two well-known traversal algorithms: breadth-first search (BFS) and depth-first search (DFS). The results are summarized in Table 7

Table 7: Performances of graph traversal algorithms

Number of vertices	DFS [ms]	BFS [ms]
100	0.019	0.008
1000	0.435	0.096
10000	8.637	1.885
40000	103.342	41.515
100000	284.324	136.483

The DFS traversal is about two times slower than the BFS traversal once we use a large enough graph. Although both algorithms have the same time complexity $O(|V| + |E|)$, in the case of our random call graphs, it appears more efficient to use BFS than DFS. But, the space complexity of BFS, $O(|V| + |E|)$, is greater than DFS, $O(|V|)$.

7.4 Getting information about the functions

The parsing of the libraries is independent from the graph size. It depends on the size of the libraries or executable files. Furthermore, the number of virtual tables and virtual functions in the ELF format also matters because it will be more costly to parse.

We separated the benchmarks in two parts; one for the executable files and one for the shared libraries. The algorithm to get the information is not the same.

We tested the old version forking to *readelf* and the last version using *libelf*.

For this benchmark, each file is parsed twice without measuring the time and then we take the mean of the 25 next executions as the time needed to parse the given library.

We selected several different executable files for this benchmark:

- clang: The CLang compiler, 39.7MB
- skype: The Skype application, 20.4 MB
- inlining: The analyzer itself, 407KB
- eddic: The EDDI compiler, 357KB
- virtual: A very little executable created for testing purpose during this project, 14KB

Table 8 presents the results of this first part of the benchmark.

Table 8: Performances of parsing an executable

Executable	readelf version [ms]	libelf version [ms]	Factor
clang	3633.870	179.963	20.192 %
skype	1375.760	5.015	274.329 %
inlining	23.940	1.720	13.919 %
eddic	23.511	2.423	9.703 %
virtual	6.796	0.322	21.106 %

We also selected several different shared libraries for the second part of the benchmark:

- libCaloCalibHitRec: One ATLAS library, 1.2MB
- libRooFitCore: An ATLAS library for matrix computations, 10.7MB
- libGaudiSvc: One library of the GAUDI framework, 6.6MB
- libdyninstAPI: The base library of Dyninst, 45.3MB

- libinstructionAPI: The library of Dyninst InstructionAPI, 3.9MB
- libclang: The library of CLang, 14MB
- libEnhancedDisassembly: Another CLang library, 22.3MB
- libLTO: A CLang library, 20.8MB
- libwireshark: The library of Wireshark, 44.4MB
- libQtWebKit: The web kit library of Qt, 21.2MB

We selected more shared libraries because ATLAS contains a lot of libraries.

The results are summarized in Table 9.

Table 9: Performances of parsing a shared library

Shared library	readelf version [ms]	libelf version [ms]	Factor
libCaloCalibHitRec	134.960	9.138	14.769%
libRooFitCore	719.153	61.305	11.731%
libGaudiSvc	672.829	49.149	13.690%
libdyninstAPI	393.535	42.411	9.279%
libinstructionAPI	26.194	2.100	12.473%
libclang	294.806	1.723	171.100%
libEnhancedDisassembly	474.755	1.146	414.271%
libLTO	407.333	1.148	354.820%
libwireshark	1072.600	12.736	84.218%
libQtWebKit	842.977	4.904	171.896%

The difference between the two algorithms is really large. The main reason being that we do not read the whole file, we read only what we need. So, if there is no virtual table, we do not read the sections containing the content of the virtual tables. We also avoid the cost of forking to another program and the cost of reading twice the library. *readelf* has to read the library to produce the output and then we read again the output as it was the shared library. Lastly, we acquire information by reading bytes by bytes, not by parsing everything into a string and using expensive string operations to find the good piece of information.

Using the first version could have been a performance bottleneck in the analyzer, but with the new version, the amount of time needed to read even a huge shared library is totally acceptable. However, in the case of an ATLAS call graph with hundreds of libraries, this operation is the slowest. We have to note that the old version of the algorithm could have been more optimized but absolutely not enough to be more efficient than the new one.

7.5 Analyze the graph

We tested the performances of the computation of the temperature for the call sites and the functions. This computation is a matter of iterating through each edge and vertex and computing the temperature. Table 10 summarizes the results of this benchmark.

Table 10: Performances of heuristic calculations

Number of vertices	Functions [ms]	Call Sites [ms]
100	0.004	0.083
1000	0.036	1.100
10000	0.298	15.119
40000	1.228	101.655
100000	3.178	298.234

The computation for the call sites is about hundred times slower even though there are ten times more call sites than functions. This is explained by the fact that the computation of the temperature for a call site is more complicated than for a function. Moreover, this is slowed by the fact that iterating through call sites is slower than through functions. But once again, the complexity of the algorithm is linear.

We also tested the performances of the different analysis algorithms.

First, the performances of the clustering algorithm have been tested. For this algorithm, the performances can depend on the input because we are searching paths between functions. That is the main reason why five different random graphs are used to test the performances of the application. Table 11 presents the results of the clustering algorithm.

Table 11: Cluster algorithm performances

Number of vertices	Time [s]
100	5.444
1000	345.896
10000	89.564
40000	490.102
100000	1372.240

The results are not increasing linearly but a bit less. This is due to the fact that the temperature has a tendency to be lower for the call sites when the size of the random graph increases. And the clustering algorithms performance depends on the number of hot call sites in the call graph.

Then, we tested the library issues search algorithm, for which you can see the results in Table 12.

Table 12: Library issues search algorithm performances

Number of vertices	Time [s]
100	12.034
1000	119.752
10000	543.164
40000	3296.600
100000	10331.500

This time, the algorithm is not absolutely linear, but has a complexity a bit higher, without being linearithmic or quadratic. If the parameters of the algorithms were higher, that will lead to an algorithm being exponential.

The circular dependencies search algorithm has also been tested, as you can see in Table 13.

Table 13: Circular dependencies search algorithm performances

Number of vertices	Time [s]
100	2.334
1000	25.938
10000	279.088
40000	1266.770
100000	3399.030

Again, the algorithm has a linear complexity.

In conclusion, none of the analysis algorithms is time consuming, even when applied on a large graph. The performance bottlenecks of the application are located on the graph reading and file parsing operations.

7.6 Memory usage

Even if it is not directly performance-related, the memory used by the analyzer is also an important factor. We measured the used memory for the graph of the application. For that test, we read the graph from the random graph and measure the memory usage directly after the reading of the graph. Table 14 presents the

Table 14: Memory usage of the graph

Number of vertices	Memory usage [MB]
100	2.867
1000	8.390
10000	63.804
40000	248.488
100000	618.082

As we measure the memory usage of the whole application, the result is not really precise for small sizes. For graphs with 10'000 vertices and more, the memory usage starts to be stable. We can estimate the memory usage to about 6.4KB for each new vertex (and its 10 call sites).

The memory usage is not negligible and grow up quickly. We can estimate that having a graph with one million nodes would require about 6GB of memory. In our case, it is not an issue because the computers where will run the analyzer have enough memory to handle such big graph, but this will perhaps not be the case of the users of the application.

If we wanted to reduce the memory consumption of the analyzer, there are some tracks that can be followed:

- reduce the number of properties. Each property of the graph takes some memory for each element. One thing we can imagine is to rewrite the DOT file reader to parse directly the labels and so to not store it in the graph because it is redundant with the properties used to store the information contained in the label. If really necessary, we could also calculate some data every time we need it like the frequencies or the number of parameters instead of storing it, but that would of course decrease the performances of the application.
- use another graph structure. The last versions of Boost Graph Library includes a new graph structure, *compressed_sparse_row_graph* that takes less memory to store the edges and vertices. The biggest difference is that this structure is immutable. We cannot add or remove any edge from the graph once it has been constructed, but we can edit the properties of each element. Once again, we would have to rewrite the DOT file reader in order to know the number of functions and call sites before creating the graph.

7.7 Tests on ATLAS

Although it is not really a benchmark, the tests on ATLAS allowed us to find performance issues and to fix some of them.

At the beginning of the tests on the ATLAS call graph, the performances were not very good. Indeed, the full analysis took about 30 minutes. At first, we thought it was because of the library parsing, but after some profiling we realized the time was spent in using the information parsed from the files. After

the parsing of every library, more than 700'000 functions were in memory and simply searching into these functions was taking most of the time. Actually, the containers were not used in an efficient way. Once we fixed this problem, a single run took only about two minutes.

We also improved the performances using hash tables instead of red-black tree to store this information. The search in hash tables can be done in amortized $O(1)$ complexity, compared to the $O(\log n)$ for the red-black-tree structure. Using these new data structures saved about a third of the run time, allowing a full analysis to only last about 80 seconds. We used the implementation of TR1 provided by gcc.

When working on the graph with demangled names, we have to demangle the name of the function taken from the library. It turned out that the demangling was more expensive than we thought. After optimizing the algorithms to use as few as possible demangling, the run-time of the application has been improved by an order of magnitude of 2.

7.8 Function call

In order to estimate the cost of a function call, we developed a small benchmark to weigh the different parameters taken into account for a function call.

For this benchmark, each function tested is invoked two million times. The elapsed time is then calculated. This process is repeated 12 times and the mean of the last 10 samples is taken as the result.

We started to bench the impact of the number of parameters for a function call. We have created several versions of the same function with a different number of parameters and invoked then directly with the same parameters value. Table 15 shows the result of this benchmark.

Table 15: Overhead of long parameters

Number of parameters	Mean [ms]	Factor
0	4428	1.00
1	4723	1.06
2	5156	1.16
3	5446	1.22
4	5828	1.31
5	6569	1.48
6	7472	1.68
7	8019	1.81
8	8610	1.94
9	9260	2.09
10	10031	2.26

The overhead of each new parameter is not negligible. If we take the mean of the differences divided by the number of parameters, we find 10% of overhead for each parameter. All the parameters are 64 bits

long so that each of them fit on a register. Moreover, this is also the size of a pointer, or a reference, so, we can easily interchange what we want in this number of parameters.

To verify if the size of the parameters matters when they fit in registers, we also made the test using int, encoded using 32 bits. The results are presented in Table 16

Table 16: Overhead of int parameters

Number of parameters	Mean [ms]	Factor
0	4428	1.00
1	4687	1.05
2	5025	1.13
3	5707	1.28
4	6194	1.39
5	7077	1.59
6	7660	1.72
7	8576	1.93
8	8739	1.97
9	8884	2.00
10	9691	2.18

The results are about the same as the results for long parameters. Copying int or long results has the same cost, so that the overhead on the function call is about the same.

More than the number of parameters, we also tested the impact of the size of parameters in the case of pass by value. The used objects are simple objects containing only an array of longs. The size is changed for every class using a greater array. Table 17 summarizes this benchmark results.

Table 17: Overhead of the parameters size

Size of the object [bytes]	Mean [ms]	Factor
0	4428	1.00
32	7188	1.62
64	10837	2.44
128	20288	4.58
256	41556	9.38
512	63623	14.36

The cost of copying parameters can be rather large. The overhead seems to be amortized. Indeed, the impact per byte is twice as small for a 512 byte parameter than for a 32 bytes. It is hard to estimate exactly the impact of the addition of a certain number of bytes, but we can conclude that having big objects passed by value is generally not a good idea if it is not absolutely necessary.

The impact of virtuality on the performances has also been tested, as you can see in Table 18.

Table 18: Overhead of the virtuality

Type of call	Mean [ms]	Factor
Normal call	4428	1.00
Virtual call	6142	1.387

The virtuality overhead is about 39%.

Finally,the overhead of calling a function in another library has been measured. The results are shown in Table 19.

Table 19: Overhead of library call

Type of call	Mean [ms]	Factor
Normal call	4428	1.00
Library call	6156	1.390

The overhead of a library call is also about 39%, so we can say that the overhead of function call and library call is about the same.

This benchmark allowed us to have a better idea of the weight of each parameter playing a role in the function call overhead. It turned out that the impact of the parameters was higher than we thought at first. That helped us create a temperature heuristic more accurate. Moreover, the fact that we developed a full benchmark for that point can help the user improving the heuristic for its system. Indeed, as the weights of the different parameters are configurable in the heuristic, the user can run the function benchmark and then adapt the values to the ones generated on its system.

8 Refactorings

This section presents the refactorings that were made during the implementation in order to improve the design or the usability of the application.

8.1 Bundled Properties

At the start of the project, we started using the properties as described in Section 3.2. This kind of properties is named Internal Properties. For every property of the edges and the vertices, you have to declare a new property type and chain it with the other properties of the edges, respectively, the vertices. And when you want to read or write a property, you have to get the good property map and then use the descriptor as a key to finally make something with the property.

But we realized that there was an easier way to manage properties. Indeed, Boost Graph Library 1.32.0 introduced a new kind of property, named the Bundled Properties. Using this system, we can specify classes or structures representing the whole information of an edge or a vertex and access the information directly. So, we do not use anymore the properties and property maps, we simply manipulate objects' data. We can get the object (named bundle in BGL) using the operator [] of the graph with the descriptor as a key. And then we can directly read and write the properties.

Using Bundled Properties instead of Internal Properties in the analyzer has allowed us to remove a lot of cumbersome code and to make the code clearer.

The only limitation of the Bundled Properties is that they require partial class template specialization and some compilers do not offer this feature although most of the recent compilers are smart enough to compile that kind of code. In our case, it is plainly supported in gcc.

We can compare both property system on a simple example, an old version of the temperature function. Listing 23 shows an implementation using Internal Properties.

Listing 23: Use of Internal Properties

```
double computeCallSiteTemperature(CallSite& site, Graph& g){
  property_map<Graph, vertex_self_cost_t>::type selfCosts = get(vertex_self_cost, g)
      ;
  property_map<Graph, edge_frequency_t>::type frequencies = get(edge_frequency, g);
  property_map<Graph, vertex_size_t>::type functionSizes = get(vertex_size, g);
  property_map<Graph, edge_temperature_t>::type temp = get(edge_temperature, g);

  Function caller = source(site, g);

  double cost = frequencies[site] * selfCosts[callee];

  temp[site] = cost / functionSizes[callee];
}
```

And Listing 24 shows an implementation that uses Bundled Properties.

Listing 24: Use of Bundled Properties

```
double computeCallSiteTemperature(CallSite& site, Graph& g){
  Function caller = source(site, g);

  double cost = g[site].frequency * g[callee].self_cost;

  g[site].temperature = cost / g[callee].size;
}
```

It is very clear that the implementation using the Bundled Properties is much simpler than the other.

8.2 Filtering

After some time, we saw that some of the candidates were located directly in the standard library and were often the same. For example, we had the call site from *malloc* to *__int_malloc* frequently reported as a good candidate to inlining.

To avoid having this kind of result too often, we added a filtering feature to the application. The filtering occurs at the very end, just to hide some results to the user. We did not make the filtering before this in order to avoid losing information. There are some default filters available in the application that the user can enable. Some functions often used like *malloc* and *free* are filtered by the these filters. The user can also directly add new filters. A function is filtered if its name is present in the filter list and a call site is filtered if its source or target function has to be filtered.

There is also another form of filtering that we added to the issues, the duplication filtering. In the case of library issues, it is possible that there are several links that can make a function interesting to be moved from a library to the other, so the user does not want to see it several times. We filter the solution so that it is not printed several times. This specific feature is also configurable by the user.

We also added a feature to specify that we cannot move functions in some libraries, typically the system libraries. We can move functions from these libraries into our libraries, but we cannot move functions into them. The user can choose which functions he wants to make as protected. The functions in a library considered protected are not directly filtered from the issues, but when an issue suggests moving a function into one of these libraries, the application outputs a warning indicating that it is not possible to move a function to this target library.

8.3 Usage of the analyzer as a library

During the project implementation, it was asked that the application should be used as a library and made available to the public. So the application has been rewritten providing clear interfaces to be used by another application. Instead of printing issues in the console, the user can now get the issues using the functions of the library. This has been achieved by separating the concepts of searching (*Analyzer*) and reporting (*Reports*), so that the user of the library can get the information he wants or directly use

the reporting system to output information in the standard output. More than rewriting the interfaces, it has been necessary to document the interfaces using Doxygen in order to give the user information regarding which classes are responsible for what and which functions to use to achieve a specific goal.

As the library will be open source, it has been necessary to put the project source code under an open source license. The chosen license is the GNU Lesser General Public License (LGPL)[24]. This license is especially designed for libraries. It does not apply restrictions on the program that merely link to the library, so the library can be used in proprietary software as long as it is not a derivate work. Applying the LGPL license means that every file has to contain the license statement as a header. Moreover, the application must be distributed with a full version of the license file.

8.4 Customization of the parameters

During the development of the different algorithms and temperature functions, more and more parameters have been added. We have decided to let the user modify them to customize the rules of the program. This adds to the library customization possibilities.

The user of the library can now programmatically modify a set of parameters used in some algorithms. More than the parameters of the different algorithms, we also gave the user the opportunity to change the different values of the temperature heuristic. It is possible that the impact of the different parameters are different from one machine to the other, from one architecture to the other and from one compiler to another. So, the user can run the function call benchmark and then adapt the values to the one that are relating to its system.

Of course, default values of these parameters are kept in the application, so that we can use the algorithms without changing anything, or by only changing some of the parameters.

Aside from adding features to the user, this refactoring has other advantages:

- no more magic numbers in the code
- all the default values are at the same place in the code
- the code is more readable without many numbers everywhere, we refer to the parameters in the code

This is done using the *Parameters* class providing static utility functions to get and set the values of the parameters. We chose to use a class with static functions to avoid having to share the instance of the parameters between every class and to avoid the use of a *singleton*.

As the the analyzer can also be used directly in command line, all the parameters do have a corresponding options to be modified from the command line. All what the tool does with the options is to parse them and to send them to the *Parameters* class.

9 Problems

This section contains the main problems encountered during this project.

9.1 Understanding the Boost Graph Library

The Boost Graph Library, like most of the Boost C++ libraries, uses a lot of template techniques in order to be as efficient and generic as possible. Templates enable the library to use metaprogramming to resolve as many things as possible directly at compile time. More than metaprogramming and generic programming, BGL also makes an extensive use of traits and concepts. The traits classes are a way to store information about compile-time entities and the concepts are a way to define requirements about template types in a formal way.

At first, all these techniques are hard to understand and to work with for an intermediate C++ developer. The code is hard to read because of the intensive use of templates and the compile time errors are even more difficult. It is not rare to find a template function name that fills half of the screen in the console. The Boost Graph Library uses compile time assertions to produce errors at compile time to detail the error, but most of the time the information is not easy to understand and is lost in the tons of compile-time errors about templates.

More than understanding the techniques, we also need to understand the code we write. It is not always easy to understand exactly what can do a specific piece of code because the usage of metaprogramming and macros are changing the code we write in something more powerful.

9.2 Creating a Boost Visitor

Several algorithms of the Boost Graph Library are based on the visitor pattern, like the depth-first or the breadth-first searches. There are some constraints on these iterators that we must consider before starting implementing a new one:

- The visitor has to be passed to the *boost::visitor()* function before being passed to the algorithm.
- The visitor is passed by value, so it must be copy convertible. This means that we must pay great attention with stateful visitors.

These constraints are simple, but can cause some troubles when we are not used to them. The most noticeable error is trying to store state inside the visitor. The visitor is copied because passed by value and then you have only a reference to the source visitor which state will never be modified. So, when we need to have variables inside visitors we must create them outside the visitor and give the visitor a reference to them so that it can edit them and produce an output.

9.3 GProf2Dot issues

GProf2Dot is the tool that converts a Callgrind profile file into a DOT file. There was a big lack in this conversion. Indeed, neither the library information nor the filename were added to the result file. We need this information for the analyzer. There were two solutions to solve this problem:

- edit the Python script
- parse the Callgrind file directly in the analyzer

After a look into the Python script, we saw that making the parsing ourselves would need a lot of time because of the complicated Callgrind format. So, we decided to edit the Python script directly. Doing that, we also chose to remove all the information that we do not need in the DOT file, for example the font color and the arrow size. We added two new properties to the vertices in order to store the new information: *module* and *filename*.

Removing the graphical information was not difficult. We have edited the *DotParser* class of the Python script to not output these properties.

Then, we edited the script in order to find the information about the module of each function and to store them in the graph. The issue was not public, but we found in the code a *FIXME* comment indicating that the extraction of the module and filename was broken and disabled. We tried to reactivate this feature, but we saw that the results were clearly broken. Each function was bound to a library, but in most cases, not the good one. The big problem to solve this issue has been to understand the Callgrind format, which is really complicated and uses a lot of compression. Another thing about the Callgrind format is that, depending on the presence of the debug symbols in the executable, the call graph is not exactly the same. It was harder to find the information about the library when the executable had been compiled using the debug symbols. So we chose to accept the limitation that the call graph must come from an executable without symbol tables. Then, we found a way to get the information. Depending if the function first appears as a callee or a caller, the information about the library is not at the same place, so we changed the script slightly to take that into account and it worked. We also had to work with the fact that the input profile can be compressed or not, so we wrote the module search for that it can work with both inputs.

Moreover, the Python script has another drawback. There were several informative nodes at the start of the graph. These nodes were here only for graphical information, so we did not need them and removed them from the digraph. If we had chosen to let these elements in the graph, we would have need to add new properties to the nodes because these special nodes have a lot of information only here.

The next problem of the Python script was that the values of the events were always converted to relative values in percentages. In our case, it could have been interesting to have the real values to not risk losing some information. Once again, to avoid wasting time on a product that we will normally not use once Gooda has been released and because the code was not easy to refactor, we did not solve this problem.

Finally, in some situations, the Python script was not able to parse the Callgrind file. We wanted to try to analyze large applications like Mozilla Firefox, but when we converted the Callgrind profile file using the Python script, it took ages to finish with a stack overflow error. Increasing the max recursion depth limit of the Python interpreter did not solve the problem.

9.4 Dyninst

The installation of Dyninst has been difficult. The library is based on a lot of other libraries, so you have to install them and it is not clearly defined what depends on what. First, because we needed only a part of Dyninst, we tried to install only these subparts. But it is not easy to find the dependencies of the sub part. So, we decided to install the whole library at once and to include the whole header files in our simple test application.

The first problem we faced during the installation was that a specific function *cplus_demangle*, normally provided by *libdemangle*, was not found at link time when we installed everything from the binary releases. There was no information in the documentation of Dyninst on how to install this library that comes normally with *bin-utils*, but not in the case of a 64 bits system or perhaps even in the case of an Ubuntu 64 bits.

Finally, we chose to build the library from the sources to be sure that the problem was not from the binary release. We had to disable the test suite compilation because it was impossible to build it, but after that, we were able to use the library.

Another problem was the lack of examples. For parseAPI, only one example was available and it was not complete; all the *include* and *using* directives were missing, so we must find out by looking in the reference documentation and in the sources which files we have to include and in which namespace are the classes we use. And for the symtabAPI, there is no complete example, we have to get parts of code in the documentation.

9.5 Virtuality

Finding a way to detect the virtuality of a function has not been as easy as we expected at first. Indeed, there is no library to compute this information from an assembly. We started with the fact that this information had to be in a shared library because this information is necessary at link-time for the application to know the functions to call.

So we searched the virtual table content in the library file. For that, we made an extensive use of the Linux utilities like *readelf* and *objdump*. There is not a lot of information available on the subject because this is compiler and architecture dependent. We finally found this information on the ELF file. More information about the extraction is available in the Section 3.4.1.

Using parseAPI, we can detect every call to a function. By doing that, we can determine that a function is not virtual because there is a static link to it, but we cannot say that the other functions are virtual or not. Indeed, this works only in a given library. In order to make this solution works well, it would

have been necessary to parse every application library, keeping in memory every static call and then we could say that the function that has not been found are either virtual or not used. Moreover, this technique is not one hundred percent accurate. For example, we may have calls using function pointers that will not be detected using this technique.

If we had not found the solution using *readelf* directly, another solution would have been to read the sources directly, but this would have been a new limitation to the analyzer that the whole sources are available to read. Furthermore, this may also have been a performance bottleneck because we would have to parse a large number of source and header files to compute this information.

Because the computer used during this project is running in 64 bits, we made all the tests in 64 bits shared library. But the ATLAS libraries are compiled using 32 bits, so that we have to make our algorithm works with both formats. You can find the format of the library or executable by reading the Class attribute of the ELF header, ELF32 indicating a 32 bits file and ELF64 a 64 bits file.

Because at first the algorithm has been tested only on some little libraries, we did not really take the performances into account. But when testing this algorithm on an ATLAS library, it took a very long time (>10s) to finish the search of the virtual functions. The first algorithm was made of four nested loops. The first step to improve its performance has been to reduce the nesting of the loops. As a matter of fact, we have reduced it to one simple loop. We can make the search a lot faster by sorting the virtual tables by their starting address before searching for the functions. Then, we have to go through every saved line only once and each time we are farther than the end of the current virtual table, we have to go to the next one until we reached the last virtual table. Using this optimization, even on the ATLAS library, the algorithm takes less than 200ms to perform the full search.

9.6 Function sizes

After choosing not to use parseAPI, we wrote the whole research directly using the content of the ELF files.

The first version worked well in a simple case, but we quickly saw that there were several drawbacks to this first version.

First of all, we considered a function to have a mangled name starting with *_ZN*, but with that limitation we lost, for example, the overloaded operators that are also considered functions. So, we chose to get the sizes of every line appearing in the symbol table with the *FUNC* type.

Another problem was that in the symbol table, several names are encoded using @@ followed by several characters, like *_ZNSsC1Ev@@GLIBCXX_3.4*. The problem was that in the Callgrind information, the part from the @@ characters were not present. So, we had to identify these functions and store them without the last information.

At the start of the implementation, we considered a function as unique given its name, but there are several cases where a function can be defined into several libraries. We have decided to store the size of the function for every library. With that, when verifying the size of a function we have to know the library and the name of the function. Temporarily, we disabled this feature because the Python script

was not able to retrieve the good libraries for each function, so that we had no result during the analysis.

9.7 Function call benchmark

To have the most realistic temperature function as possible, we developed a benchmark to weight the different parameters playing a role in function call.

But it is not that easy to have a benchmark with realistic results. The compiler is doing a lot of optimizations, so it is highly likely that the function we want to test will be inlined or the parameters passed to the function will be ignored if they are not used.

To solve a part of the problem, the function declarations have been put in a header file and defined in another source file. This way, the code invoking the function does not know the implementation and so the call cannot be inlined.

Moreover, when benchmarking a very small part of code, like a single function call in our case, the results can be very instable because we are benchmarking only a few CPU instructions. It is very important to run the benchmark in the most isolated possible environment in order to avoid interruptions that can largely change the results of the benchmark.

9.8 Multiprocess application

When we profile an application that launches other processes, we face a problem when using Callgrind. By default, Valgrind does not profile the child processes. To fix that issue, we have to use the *–trace-children=yes* option of Valgrind to check the sub processes. Even with this option, we have another problem. Valgrind checks each process separately, so we have a Callgrind profile for each created process.

In the case of CLang, the problem was that the C++ compiler forked each file to compile to clang, so we had only little profiles to analyze. If we want more accurate data, we have to compile bigger files. There is no option in Callgrind to merge several profiles into one. The temporary solution we found in the case of CLang was to use a so-called "Unity Build". We created a new C++ source file including all the others source files and we compiled this source file. This is a technique used to decrease the compilation time (in our case, the compilation time has been decreased by a factor of 2). With that, we have a more complete profile for our compilation, but the resulting application can be different because the compiler have a better view of the whole application.

The case of multi-threaded applications is not a problem. By default, CallGrind profiles all the threads of an application together. You can change this behavior separating each thread into its own profile. That can be useful in some cases, but in our case, we always wanted to profile the whole application.

9.9 Tests on CLang

It has not been easy to implement the issues proposed by the tool. Inlining a single call site is not always possible without modifying some code. Some functions use static variables not accessible from the call site and then we need to change the function to inline it, possibly with a lack of performances in the case where it is not inlined. On the contrary, sometimes we had to augment the visibility of some fields in order to inline a call site that uses them and that new visibility can perhaps change the way the compiler will optimize your code.

Inlining a whole function is also not always easy. Actually, sometimes the functions can be inlined, but are not because the compiler decided it is not worth inlining them. In these cases, we have to find a way to force the compiler to inline them. There are also functions that cannot be inlined, for example, a recursive function. And some are difficult to inline because of complicated control flows, multiple returns, etc.

Since we do not know the source code of CLang well, it is possible that some changes have implied other problems and that the saving or the loss of performances are a result of these problems and not of our changes.

Although CLang is faster and easier to build than ATLAS, it takes a long time and it is not a light program. A full build of CLang, even with several threads, takes about fifteen minutes. Moreover, a full CLang build takes a lot of place on the hard disk. On the computer used to make the tests, all the versions of CLang together are using more than 8GB.

Another problem we have faced is the templates. The templates are instantiated at compile time and then are not present in the assembly. So when we saw a call site in the call graph that comes from X, it is possible that the call site is declared in the template, then compiled into X. So, to not inline too many calls, we have to create another template in which we inline the incriminated call site.

Of course, the compiler is doing a lot of optimizations in order to produce efficient algorithms so that the assembly does not always reflect the sources. The most noticeable modification is of course the inlining. In the source, there may be a path where X calls Y and Y calls Z, but in the assembly the call to Z has been inlined and then, there is only a call from X to Y. When we find a call site in the graph that is not in the source, we have to follow all the paths from the source function to find a call to the target function and this can take time regarding the function complexities.

Finally, there are sometimes several parallel call sites. If the analyzer proposes to inline a call site from X to Y and there are several similar call sites, we have no ways to differentiate the call sites. In this case, we chose to inline every call site from X to Y when the application proposes this.

In the case of CLang, the assertions helped a lot. There are a lot of asserts in the whole application so that we are warned if we broke something. Of course, the asserts cannot cover every problem we could have inserted, but it is a good security.

9.10 Tests on ATLAS

Testing the analyzer on ATLAS has not been a easy task. To avoid wasting time on installing everything, the tests have been made on a remote computer located at CERN, in Switzerland. This situation led to some problems because this computer does not have the same configuration as the machine used to develop the application.

The first problem we had is that the *libelf* library installed on the computer was not exactly the same and one specific function was missing, being replaced by another. Where the normal function followed the common convention of returning zero on success and another number on problem, the new function returned one on success and zero if there was an error. So we had to use preprocessor instructions in order to use the good version for each *libelf* implementation. So the program has to be compiled with flags to compile and works well on the machine.

Another problem was that the provided call graph was an old one and that some libraries have been moved from this time, so that we had to edit the call graph in order to find the good libraries.

Also, since the call graph predated the project, it was not following the limitations we had emitted about the Python script and on the application. First, the graph was compressed and the edited Python was no longer supporting the compression. We had to correct the script to support the compression. Then, the call graph contained demangled names and that was not supported by the application. We had to make several changes to support the demangled names, especially demangling the names from the shared objects to bind them to the ones from the call graph.

10 Tools

To develop this application, we have used only a few tools. Indeed, in the C++ world on Linux, there is no outstanding editor and most of the developers are using command-line tools and basic text editors to develop C/C++ applications. In our case, we used Vim as the editor to develop the library.

At the beginning of the project, we used *make* to build the project. But we switched to CMake[25] after some weeks. CMake is not really a build automation tool, it is a generator for other tools. In our case, we use it as a makefiles generator. So, we have to launch CMake to create the makefiles and then *make* to build the project. The advantage is that the CMake configuration file is easier and more expressive than a makefile. Moreover, CMake manages the dependencies between the files, so that the generated makefile is very complete. Finally, it is also an advantage when we want to port an application on other platforms because CMake can also generate Microsoft Visual C++ project files, so you can build it on Windows as well.

Another tool that has been really useful during this project is c++filt. This little tool demangles all the C++ mangled names directly passed to it or on the standard input. It has been useful when working on the tests, in order to easily find where a function was in the source code without losing time working with the mangled name.

We used Doxygen[26] to document the source code of the application. Doxygen is one of the most used documentation tool for C/C++ projects in the Linux world. It enables us to describe every part of the public interface and then to generate complete HTML documentation of the project. You can also generate documentation in other formats like RTF, LaTeX or Unix man pages.

Although there is no outstanding editor, there are a lot of command-line tools that can really help a C++ developer to produce robust applications. In this project, we have used several of these command-line tools. We used cppcheck[27] as a static analyzer. This open source application checks your code to find allocations errors, variable issues, members that can be *const* and other defects. We also used cpplint[28] that is a simple Python script checking your code base in order to validate the code style according to the Google C++ Style Guide. More than testing the code for the style, it can also point you to some programming mistakes. And last but not least, in order to have a clean code and with the same style everywhere, we used AStyle[29]. This tool formats your entire code according to rules you can give to it. The use of all these tools enabled us to solve some problems, to remove some unused variables and to have a cleaner code base.

More than being helpful to generate the call graph, we also used Valgrind to check for memory problems in the application. We have corrected some memory leaks in the application and some problems about uninitialized values. Furthermore, we used Callgrind to profile the analyzer to find performance bottlenecks when working on the ATLAS call graph.

11 Conclusion

During the benchmarks, the analyzer has shown that it can produce some good results. Inlining only few call sites or functions has quickly given us some savings greater than 1% when testing our application on the CLang Compiler. The tool can to be used when the other optimizations have been made, but it can also shows to the developer where the hot regions of the code are located. For the ATLAS software, saving 1% of the run-time saves about 100K$ per year for the experiment.

A goal of the project was to understand if it was worth to consider the clusters of hot call sites when choosing inlining candidates. Our results demonstrated that it has advantages to inline a cluster of the same temperature as a single call site. Nevertheless, if we compare the number of call sites to inline, it is better to inline single call site considering that it is less work to inline it by hand. It would be interesting to test the effects when doing profile-guided optimization in a compiler.

At the beginning of the project I thought that the longest part of the project would be to work on the temperature heuristics and to test the application, but the information extraction phase has taken a lot of time. Indeed, I assumed that the available tools would give us everything we needed. I turned out that it will be up to the analyzer to directly extract the information about the functions. For that, I had to study the ELF format of the shared libraries and executable files and then extract the data from the analyzer.

Furthermore, we thought that the most interesting report that the application could produce would be a list of functions and call sites to inline. It turned out that with the available information we have been able to detect other issues. The analyzer is able to detect issues between the libraries or hierarchies of virtual functions that could be improved.

Testing the code has also been harder than I thought. We are not working at a compiler level, so the issues provided by the analyzer have to be resolved by hand. When a developer analyzes its own program it is easier because he knows its code, but when we have to edit another program, it becomes more difficult. Moreover, this can lead to errors. We can optimize more than we wanted or, on the contrary, introduce a new performance issue.

I hope that this project will go further and that it will help the ATLAS Software team to improve the performance of their applications.

11.1 Future work

Because the Gooda tool has not been ready before the end of the project, the analyzer has not been tested on its generated call graphs. Once Gooda will be finalized, it will be necessary to test the analyzer on its output to see if the two call graphs are compatible. Moreover, it is possible that the Gooda call graph contains more information than Callgrind and that the analyzer can take advantage of them. It is even possible that some information from the generated spreadsheets can be extracted. If the use of Gooda call graphs is successful, it will allow the removal of the dependency to the slow and not-well-working Python script and the slow Callgrind profiler.

Until Gooda is finished it could be interesting to support others profilers. The Python script is not only made for Callgrind: it supports several other profilers. Not all of them would be interesting because for some of them information is missing. For example, we tried *gprof*, The GNU Profiler, and it did not get the information about the shared objects containing the functions. Moreover, when the Python script on a *gprof* profile do not have the same output as with a Callgrind profile so supporting new profilers would mean doing more modifications to the Python script in order to be supported by the analyzer.

A good way to improve the analyzer would be to get the parameters size for each function when the parameters are passed by value. This would improve the computation of the cost of a function call and by the way the function and call site temperatures. This could certainly be achieved by looking to the ELF files and get the sizes of each type. For the built-in types, it would be necessary to get a map indicating what is built-in and the size of each type. The size of a standard type can depend on the compiler and on the platform. As a little improvement to the function cost model, we could also consider the function return type. Non-void functions are generally more costly than void functions. Considering the return type would then be good in that a copy of the return value can also occur and add cost to the function call. This information is not available from the mangled name so we will have to parse the sources or the assembly to discover the return type of a function.

Analyzing the source code or the assembly code can give us a lot of information that can be used when computing the heuristics. Section 2.4 detailed every parameter that plays a role when we inline a call site, but we did not take into account most of them like the optimization savings, the register pressure or the Instruction-Cache misses. Although it is not trivial, some papers have shown that they can be considered. For example, the Instruction Cache misses can be modeled[30] in order to avoid them when choosing candidates to inline. Others have shown that some predictions can be made about the effects of optimizations on a function[31]. Moreover, as perf records the events about cache misses, we can imagine to get the information directly from the Gooda call graph.

Another interesting improvement could be to remove the false positives in the virtual call search, as explained in Section 3.4.1. If we have more information about the call sites using the perf tools and Gooda, it would be possible to have the position of the call sites in the functions and then from there detect how the calls have been made. As Gooda also manages source views of the functions, it would certainly be possible to achieve this goal without too much work.

If the analyzer had to be used on projects bigger than ATLAS, the performances of the tool can become a problem. A good track would be to parallelize part of the application. On the ATLAS call graph there are two operations that take a long time, the reading of the graph and then the parsing of the information and libraries. The first point will not be easy to parallelize because it reads a single file and creates a single graph. The second point is much more interesting. The parsing of a library does not depend on other data. So, we can imagine to parse the libraries in parallel in several processes depending on the number of available processors. Once all the threads are completed, we can then get the information for each function. For the algorithms iterating through the vertices or the edges, we can also imagine splitting the iterations through several process each having only a part of the graph to manage. But the simple operations iterating through the graph are not very expensive and so they are not the best candidates to optimize. Of course, that would mean to ensure thread safety on the part

of the code that would be accessed by multiple threads. Ensuring thread safety is a complicated and time-consuming work.

During this project, we did not have time to test the analyzer extensively. We only tested the analyzer extensively on CLang. Even if CLang is a good subject when talking about large scale object-oriented applications, it would have been interesting to make more tests on other applications of the same type and also on applications of other types. It would also have been interesting to test several heuristic versions to see what we can do to improve it. We tested the tool on ATLAS, but we did not implement the solutions proposed by the analyzer in the ATLAS source code. As the analyzer has been designed for this kind of application, it would be a good test to verify its results on ATLAS performances.

So far, the output and the configuration of the analyzer is quite basic. All the information is printed into the standard output and if the profiled application is large, there are thousands of line printed to the console. A good way to improve that would be to let the user choose exactly which statistic report he wants to print and which issues he wants to be warned of. Another good improvement at this level would be to produce reports in other forms. The most practical would be to generate an HTML page, or even website, containing all the information about the profiled application. The development team could so have access to the information without launching the analysis on their own computers. Finally, it would be interesting to improve the filters of the results to allow the use of regular expressions. At that point, the filters are only checked with equality. For example, if we want to filter all the *malloc* functions, we have to put filters for *malloc*, *_int_malloc*, *_aligned_malloc*, ... Allowing regular expressions can give the user a new level of flexibility, letting him filtering all the functions of a given class for example.

11.2 What I learned

During this project I have learned a lot of things. This was my first important project in C++, so I took the chance to learn this language as much a possible by reading books and papers and applying my new knowledge on the project. I also had the opportunity to study the planned new C++ standard, C++Ox that will add a lot of new features to the core languages and to the standard library. More than just study some of the features, I tested some of them, like the very efficient hash tables (*unordered_set* and *unordered_map*) or the little, but very practical change of the new right angle bracket facility.

Of course, this was also a perfect opportunity to learn the tool chain to build C++ projects. I now know how to build a C++ project correctly, how to use shared libraries, how to create new libraries and how to build external libraries. When using C++ libraries, the best way is often to build the whole library from scratch. This may take some time and, moreover, we have to learn how to interpret the problems that may happen during these installations and fix them.

Before the beginning of the project, I was not used to the tool chain associated with C++. I think I now have a better understanding of these tools especially gcc, make and CMake. Furthermore, I have learned to develop without using a big Integrated Development Environment as I was familiar to when programming in Java with IntelliJ Idea. During this project, I have used mostly command-line tools like Vim to work on this project. This makes a huge difference, but after some time working with these tools, we have about the same productivity as when using an advanced editor. Working with the command

line tools improved also my knowledge of shell, especially bash.

This project has also introduced me to several libraries: Dyninst parseAPI and symtabAPI and especially the Boost Graph Library. It was an opportunity to study the Boost libraries set. These libraries are using a lot of advanced C++ features, so I had to understand these techniques in order to use the library in the best way. I think I have now better knowledge in capabilities like templates, generic programming and metaprogramming. I also have experienced working with libelf that is not an easy library to starts with.

Finally, I have acquired many new skills in the domain of inlining and of performances in general. I think I have good knowledge about how a function is invoked in C++ when using GCC and what factors can influence the cost of this invocation. I also have new expertise in how GCC is managing the functions and how its inlining optimizations are working. I can now understand how are compiled the executable files and shared libraries in ELF format.

12 Appendices

12.1 Appendix A: Content of the CD-Rom

In appendices of this document, you will find a CD with this content:

- This document in PDF format: report.pdf
- The document of requirements in PDF format: requirements.pdf
- The logbook in PDF format: logbook.pdf
- The user documentation in PDF format: user_doc.pdf
- *minutes*: The minutes of the project, in PDF format
- *week-reports*: The week reports, in PDF format
- *sources*: The LaTeX sources of every document
  - *report*: The LaTeX sources of this document
  - *logbook*: The LaTeX sources of the logbook
  - *user_doc*: The LaTeX sources of the user documentation
  - *requirements*: The LaTeX sources of the document of requirements
  - *minutes*: The LaTeX sources of all the minutes of the project
  - *week-reports*: The LaTeX sources of all the week reports of the project
- *analyzer*: The C++ sources of the application, the configuration files and the included tools
  - *src*: The C++ source files
  - *include*: The C++ header files
  - *tools*: The Python script to convert Callgrind profile into a DOT file

12.2 Appendix B: CLang analysis

Table 20 shows the hottest functions.

Table 20: Top 10 function by temperature

Function	Temperature
_ZNK5clang8QualType12getCommonPtrEv	0.0750
_ZNK5clang13SourceManager12getSLocEntryEjPb	0.0714
_ZN4llvm21llvm_is_multithreadedEv	0.0284
_ZNK5clang5Token17getIdentifierInfoEv	0.0264
_ZN5clang4Decl19castFromDeclContextEPKNS_11DeclContextE	0.0147
_ZN5clang8QualTypeC1EPKNS_4TypeEj	0.0144
_ZN5clang11DeclContext17getPrimaryContextEv	0.01206
_ZN4llvm16FoldingSetNodeID10AddPointerEPKv	0.01125
_ZNK5clang10TokenLexer25getMacroExpansionLocationENS_14SourceLocationE	0.0108
_ZNK5clang5Lexer17getSourceLocationEPKcj	0.0094

Then, Table 21 summarizes the first call sites.

Table 21: Top 10 call sites by temperature

Call site	Temperature
_ZNK5clang13SourceManager13getFileIDSlowEj → _ZNK5clang13SourceManager12getSLocEntryEjPb	0.0594
_ZN5clang10TokenLexer3LexERNS_5TokenE'2 → _ZNK5clang5Token17getIdentifierInfoEv	0.0235
_ZNK4llvm12PassRegistry11getPassInfoEPKv → _ZN4llvm21llvm_is_multithreadedEv	0.0141
_ZNK5clang10TokenLexer25getMacroExpansionLocationENS_14SourceLocationE → _ZNK5clang13SourceManager12getSLocEntryEjPb	0.0111
_ZN5clang4Decl22getTranslationUnitDeclEv → _ZN5clang4Decl19castFromDeclContextEPKNS_11DeclContextE	0.0072
_ZN5clang10TokenLexer3LexERNS_5TokenE'2 → _ZNK5clang10TokenLexer25getMacroExpansionLocationENS_14SourceLocationE	0.0067
_ZN5clang13SourceManager30createMacroArgInstantiationLocENS_14SourceLocationES1_j → _ZN5clang13SourceManager26createInstantiationLocImplERKNS_6SrcMgr17InstantiationInfoEjjj	0.0062
_ZN5clang7CanQualINS_4TypeEE12CreateUnsafeENS_8QualTypeE → _ZN5clang8QualTypeC1EPKNS_4TypeEj	0.0061
_ZN4llvm16BumpPtrAllocator8AllocateEmm → _ZN4llvm16BumpPtrAllocator8AlignPtrEPcm	0.0060
_ZN12_GLOBAL__N_120TemplateInstantiator29TransformTemplateTypeParmTypeERN5clang14TypeLocBuilderENS1_23TemplateTypeParmTypeLocE → _ZNK5clang8QualType12getCommonPtrEv	0.0050

The first cluster has a temperature of 0.1232 and is composed of the ten given call sites:

- _ZN5clang13SourceManager22createInstantiationLocENS_14SourceLocationES1_S1_jjj
  → _ZN5clang13SourceManager26createInstantiationLocImplERKNS_6SrcMgr17InstantiationInfoEjjj
- _ZN5clang12Preprocessor29HandleMacroExpandedIdentifierERNS_5TokenEPNS_9MacroInfoE'2
  → _ZN5clang12Preprocessor19isNextPPTokenLParenEv
- _ZN5clang12Preprocessor19isNextPPTokenLParenEv → _ZNK5clang10TokenLexer17isNextTokenLParenEv
- _ZN5clang10TokenLexer23ExpandFunctionArgumentsEv'2
  → _ZN5clang13SourceManager30createMacroArgInstantiationLocENS_14SourceLocationES1_j
- _ZN5clang12Preprocessor16HandleIdentifierERNS_5TokenE'2
  → _ZNK5clang12Preprocessor15getInfoForMacroEPNS_14IdentifierInfoE
- _ZN5clang13SourceManager30createMacroArgInstantiationLocENS_14SourceLocationES1_j
  → _ZN5clang13SourceManager26createInstantiationLocImplERKNS_6SrcMgr17InstantiationInfoEjjj
- _ZN5clang10TokenLexer3LexERNS_5TokenE'2
  → _ZNK5clang10TokenLexer25getMacroExpansionLocationENS_14SourceLocationE
- _ZNK5clang10TokenLexer25getMacroExpansionLocationENS_14SourceLocationE
  → _ZNK5clang13SourceManager12getSLocEntryEjPb
- _ZN5clang10TokenLexer3LexERNS_5TokenE'2 → _ZNK5clang5Token17getIdentifierInfoEv
- _ZNK5clang13SourceManager13getFileIDSlowEj → _ZNK5clang13SourceManager12getSLocEntryEjPb

And the second cluster has a temperature of 0.0439 and contains the eight following call sites:

- _ZN4llvm12LeakDetector20addGarbageObjectImplEPv → _ZN4llvm21llvm_is_multithreadedEv
- _ZN4llvm12LeakDetector23removeGarbageObjectImplEPv → _ZN4llvm21llvm_is_multithreadedEv
- _ZNK4llvm12PassRegistry11getPassInfoEPKv → _ZN4llvm21llvm_is_multithreadedEv
- _ZNK4llvm12PassRegistry11getPassInfoEPKv → _ZNK4llvm12PassRegistry7getImplEv
- _ZN4llvm17PMTopLevelManager16findAnalysisPassEPKv → _ZNK4llvm12PassRegistry11getPassInfoEPKv
- _ZN4llvm17PMTopLevelManager16findAnalysisPassEPKv → _ZNK4llvm4Pass9getPassIDEv
- _ZN4llvm17PMTopLevelManager16findAnalysisPassEPKv → _ZN4llvm12PassRegistry15getPassRegistryEv
- _ZN4llvm12PassRegistry15getPassRegistryEv → _ZN4llvm21llvm_is_multithreadedEv

References

[1] Gaudi's Project Website, http://proj-gaudi.web.cern.ch/proj-gaudi/

[2] GNU Compiler Collection, http://gcc.gnu.org/

[3] Scheiffler, Robert W.: An analysis of inline substitution for a structured programming language. Communication of the ACM 20(9), 647-654 (1977)

[4] Callgrind: a call-graph generating cache and branch prediction profiler, http://valgrind.org/docs/manual/cl-manual.html

[5] Valgrind: Tool for memory debugging, memory leak detection, and profiling, http://valgrind.org/

[6] KCachegrind: Profile data visualizer, http://kcachegrind.sourceforge.net/html/Home.html

[7] Gprof2Dot: Profiling output converter, http://code.google.com/p/jrfonseca/wiki/Gprof2Dot

[8] dot: Command-line tool to generate an image of a directed graph in a dot format, http://www.graphviz.org/pdf/dotguide.pdf

[9] Graphviz: Open source graph visualization software, http://www.graphviz.org/

[10] GCC Manual: 6.39 An Inline Function is As Fast As a Macro, http://gcc.gnu.org/onlinedocs/gcc/Inline.html

[11] Boost Graph Library: http://www.boost.org/doc/libs/1_47_0/libs/graph/doc/index.html

[12] LEMON: lemon.cs.elte.hu

[13] igraph: http://igraph.sourceforge.net/

[14] OGDF, Open Graph Drawing Framework: http://www.ogdf.net/

[15] LEDA: http://www.algorithmic-solutions.com/leda/

[16] agraph: http://www.graphviz.org/Documentation/Agraph.pdf

[17] NGraph http://math.nist.gov/ RPozo/ngraph/index.html

[18] Introduction to LEMON: http://lemon.cs.elte.hu/pub/doc/lemon-intro-presentation.pdf

[19] Dyninst: An API for runtime code generation, http://www.dyninst.org/

[20] Robert E. Tarjan: Depth first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146-160, 1972

[21] CLang, a C Language Family Front End for LLVM: http://clang.llvm.org/

[22] LLVM, Low Level Virtual Machine: http://llvm.org/

[23] EDDI compiler, open source compiler of the basic EDDI language: https://github.com/wichtounet/eddic

[24] GNU Lesser General Public License: http://www.gnu.org/licenses/lgpl.html

[25] CMake: the cross-platform, open-source build system, http://www.cmake.org/

[26] Doxygen, a documentation system for several languages: http://www.stack.nl/ dimitri/doxygen/

[27] cppcheck: A tool for static C/C++ code analysis, http://cppcheck.sourceforge.net/

[28] cpplint: A tool to check style guide compliance, http://code.google.com/p/google-styleguide/

[29] AStyle: A Free, Fast and Small Automatic Formatter for C, C++, C#, and Java Source Code, http://astyle.sourceforge.net/

[30] Scott McFarling: Procedure Merging with Instruction Caches. Proceedings of the ACM SIGPLAN '91

[31] J. Eugene Ball: Predicting the effects of optimization on a procedure body. Proceedings of the ACM SIGPLAN '79

Index

Glossary

breadth-first search graph search algorithm that explores first all the neighbors nodes before considering the children nodes. 51

call graph Graph containing all the function calls made during an execution of a program. 6

depth-first search graph search algorithm that explores as far as possible along each branch before backtracking to consider the neighbors. 51

DOT plain-text graph description language. It is the most used format to store graph. . 7, 9

embedded systems a computer system designed to do one specific function embedded in a complete device. 17

header only a library in which all macros, functions and classes are defined in header files. This kind of library do not need to be compiled apart. 22

heuristic technique designed to solve a problem that ignores whether the solution can be proven to be correct. 7

Instruction Cache cache for the instructions, generally put in the L1 cache of the processor, a very fast hardware cache directly on the processor. 17

JSon JavaScript Object Notation, a lightweight computer data interchange format. 7

NDA Non-disclosure agreement, legal contract between two or more parts that outlines confidential material. 11

NP-Complete Nondeterministic polynomial time complete. Solutions to this kind of problem can be verified. 7

pipelining in modern CPUs, instructions are executed in several stages and each of these stages is made by a specific component and then passed to the next one, so the CPU executes several instructions at the same, one per pipelining stage. 17

Tesla unit of magnetic field. 6

TeV Tera electron volt, unit of energy. 6

variadic function a function taking a variable number of parameters. 13, 18

visitor a design pattern allowing the user to execute operations at specific steps of an algorithm. 63

Listings

List of Figures

List of Tables