FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Ladislav Láska

Scalable link-time optimization

Computer Science Institute of Charles University

Supervisor of the master thesis:	Mgr. Jan Hubička Ph.D.
Study programme:	Informatics
Study branch:	Discrete Models and Algorithms

Prague 2017

Title: Scalable link-time optimization

Author: Ladislav Láska

Institute: Computer Science Institute of Charles University

Supervisor: Mgr. Jan Hubička Ph.D., Computer Science Institute of Charles University

Abstract: Both major open-source compilers, GCC and LLVM, have a mature link-time optimization framework usable on most current programs. They are however not free from many performance issues, which prevent them to perform certain analyses and optimizations. We analyze bottlenecks and identify multiple places for improvement, focusing on improving interprocedural points-to analysis. For this purpose, we design a new data structure derived from Bloom filters and use it to significantly improve performance and memory consumption of link-time optimization.

Keywords: compiler, link-time optimization, points-to analysis, data structures

Název práce: Škálovatelná optimalizace celých programů

Autor: Ladislav Láska

Ústav: Informatický Ústav Univerzity Karlovy

Vedoucí diplomové práce: Mgr. Jan Hubička Ph.D., Informatický Ústav Univerzity Karlovy

Abstrakt: Oba vedoucí open-source překladače, GCC a LLVM, mají vyspělé optimizéry celých programů, použitelné pro většinu současného softwaru. Stále však trpní mnoha problémy s výkonem, což zapřičiňuje nemožnost použít některé analýzy a optimalizace. V této práci analyzujeme problémová místa a identifikujeme několik kandidátů na vylepšení. Pro tento účel vyvineme novou datovou struktur založenou na Bloomových filtrech, díky které docílíme výrazného zlepšení časové i paměťové náročnosti během optimalizace celých programů.

Klíčová slova: překladač, optimalizace celých programů, points-to analýza, datové struktury

I hereby give my thanks to my supervisor, Jan Hubička, for his valuable input, helpful advice and language corrections. In addition, I thank my family and friends, who supported me during my studies.

Contents

Preface

As soon as programs started their growth, it became necessity to split them into functions, and later into compilation units. This shields the programmer from unnecessary technical details of implementation, and allows him to concentrate on the actual work. The same structure limits the scope of the compiler, which can not optimize beyond unit boundaries. This is a major disadvantage as the compiler needs as much information as it can get to generate better code.

For a long time, most compilers worked on separate compilation units, and did not really care about other units in terms of analysis. However compilers grow more capable each year, as does the computing power of consumer-grade machines. We have reached the point where advanced optimizations could be performed even on very large programs.

In 2009, the GCC (GNU Compiler Collection) project merged link-time optimizer, which enables analysis and optimization on scope of whole program or shared library. This offers new opportunities for improvement, but also new challenges. Some of the existing algorithms can easily process whole programs, some have limitations, and some of them are just too slow and/or memory intesive to be used in production.

The goal of this thesis is to explore current link-time optimization techniques, identify bottlenecks, and improve upon it.

This thesis is organized as follows: We introduce compilation and link-time optimization techniques in the first chapter. We focus on Alias Analysis problem and its possible solutions in the second chapter, with emphasis on practical use in current compilers. We improve upon Andersen's inclusion-based algorithm in the third chapter using efficient data-structure derived from Bloom filters, sacrificing some precision for tractability, and compare the results with non-approximate solution using the same algorithm. In the last chapter we provide documentation necessary to reproduce presented results.

1. Compilation and optimization

In this chapter, we will discuss the composition of modern programs, organization and size of their code-base. We proceed with an overview of compilation stages, introduce optimization passses and link-time optimization framework.

1.1 Code-base organization and size

Let us start by examining some of the code-bases for programs we use every day. Many developers run Linux, Firefox (or other browser) and GCC, but unless they are developing one of them, they do not really have a good idea of how large they are. The chart in Figure 1.1 shows historical development of code-base size over the past 10 years for selected projects.

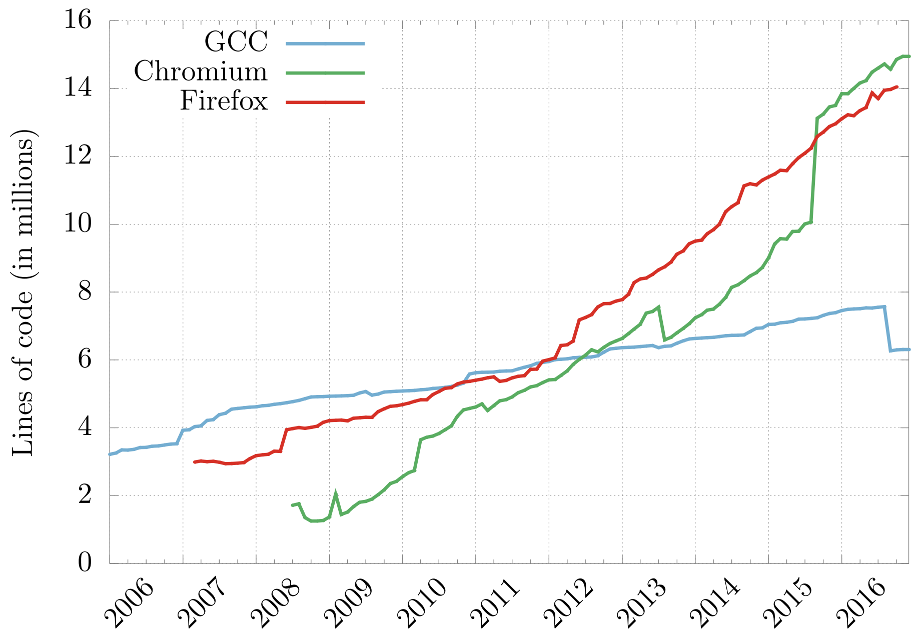


Figure 1.1: Codebase size of Firefox, Chrome and GCC over time [OpenHub]
.

It might be tempting to say that the code will be split into many libraries. It is however a common practice to bundle many libraries into a single large binary. Figure 1.2 shows 8 largest libraries contained in a standard Firefox distribution. The size of main library `libxul.so` is 66.39 MB, while the second largest is only 1.51 MB. This is due to performance optimizations, as the developers noticed a significant start-up cost of dynamic linking, and bundled them together into a single library. This also means that link-time optimizing compiler has to deal with an enormous code base at once.

We should keep these numbers in mind while designing a compiler. Compiler has to keep up with the code-base growth of projects, adding around 2 millions of lines of code each year and growing complexity of abstraction in modern programming languages.

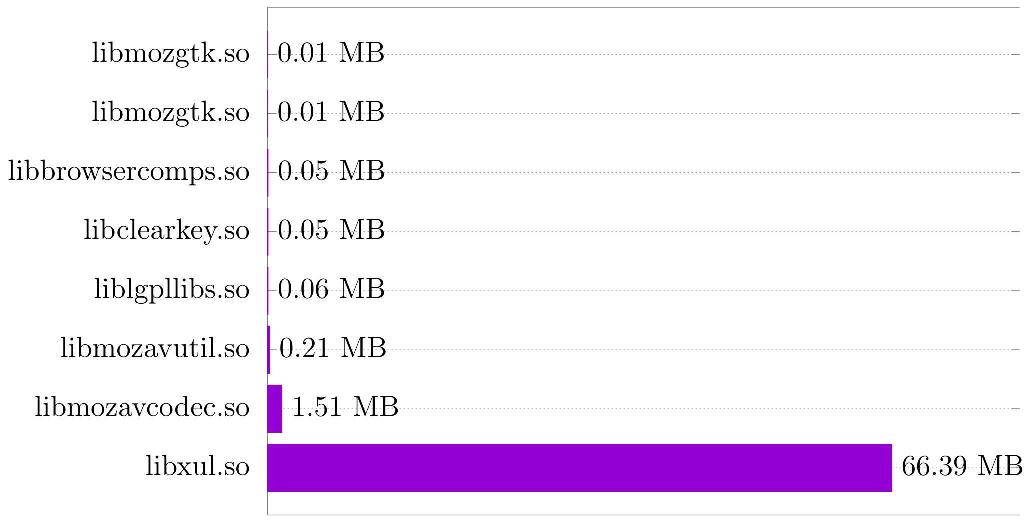


Figure 1.2: Firefox 50.0.2 object sizes by binary

1.2 Program compilation

Only the simplest programs consist of a single source file. Many programs have tens, hundreds or even thousands of source files. This not only serves an organizational purpose, but also allows the programmer to choose different optimization flags for different files, or even write different parts in different languages. A mechanism called *separate compilation* is used to compile and combine (link) all of them together, to form a finished program. Figure 1.3 shows the transformation using standard GCC and Binutils (compiler and linker).

In this traditional model, first step is to compile every source file (compilation unit) into an object file. In this phase, a compiler is invoked and does all the work necessary to convert source code into binary, including code generation. The result is stored in an object file, including required metadata, for example, symbol table. This step is independent for each source file, so they can be processed in parallel.

Second step consists of linking. The linker inspects all generated objects, resolves required and provided symbols, dynamic libraris, and produces a final executable. The linker usually does only minimal modifications to the code in object files, as it only understands symbols and sections.

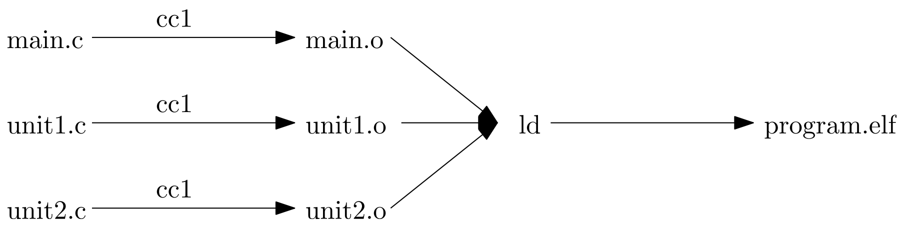


Figure 1.3: Standard workflow for separate compilation

1.3 High-level structure of a compiler

To compile a program written in a high-level programming language into a machine code requires many steps. To make orientation easy, and to support code re-usability, compilers usually consists of the following high-level components:

Front End parses the input language, builts abstract syntax tree and converts it into an intermediate language (IL) which is used by the middle-end.

Middle End analyses the program represented in IL and performs most high-level optimizations. This includes splitting the code into basic blocks, building a callgraph and control flow graph. Various optimizations are then performed.

Back End converts IL code into machine code, optimizing on the lowest level. For example, it is able to schedule individual instructions and allocate registers.

During this process multiple intermediary languages are used, sometimes more at the same time (usually during transition to the lower-level language). For example, GCC uses the following intermediate languages [GCCInt]:

GENERIC is the highest-level IL used by the Front End, able to represent syntax trees and language-specific features.

GIMPLE is tuple-based IL language, able to represent only simple expressions common to all languages. It is unable to represent many high-level constructs, for example, loops.

RTL (Register Transfer Language) is a low level language similar to machine code, containing algebraicly described instructions, as should be generated.

1.4 Link-time optimization

Restricting the scope of optimization to a single compilation unit is an important limitation to the optimizer. One example is devirtualization in C++, which needs to build complete type inheritance graph to decide which virtual function should be used to devirtualize given call. Literature usually assumes the compiler sees the whole source, but as the class its descendants are usually in a separate file, the compiler has no way of knowing that there is only one possible virtual method, and devirtualize it.

The idea of interprocedural and link-time optimization (LTO) is old. It was already discussed in literature in 1970s [All75; AC76]. One of the first industrial strength implementation of LTO optimizing compiler was MIPSPro, which open-sourced in 2000 as Open64 under GNU GPL. The compiler suite LLVM supports link-time optimization by design, from its first release in 2002 [Lat02]. GCC was entering the link-time optimization game relatively late, with the framework proposed in 2005 [GCC05; Bri+07] and released in 2009. At present day all three compiler suites have link-time optimization frameworks ready for production use.

Even before the advent of link-time optimizations, some developers worked around this limitation. For example, SQLite or older versions of KDE support code concatenation in their build system. This results in one huge source file being passed to the compiler. The result was good in its day, but still had some issues. All the code needed to be parsed at once, which increases memory usage and does not scale well, as traditional compilers are single-threaded, and thus cannot make use of multi-processor and distributed build environment.

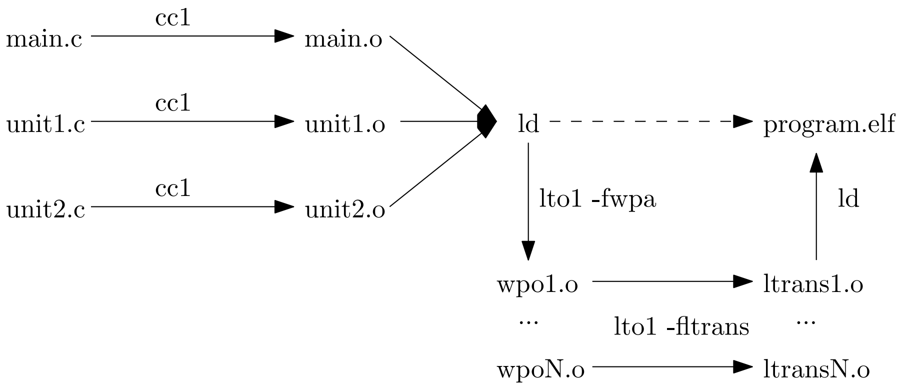


Figure 1.4: Compiling source code into binary

The LTO framework implemented in GCC (see Figure 1.4) solves these problems by keeping separate compilation, but instead of generating classical object files containing machine code, the middle end stops and writes GIMPLE representation into the object, including some metadata (for example the call graph).

Instead of generating library, the linker then picks up the GIMPLE representation and invokes the compiler again. GCC was designed to perform most of the link-time optimizations in parallel, and the process has been further split into two stages. The sequential WPA (Whole Program Analysis) stage and parallel LTRANS (Local TRANSformations) stage.

WPA stage performs declaration and type linking, and decision stage of inter-prodecural optimizations. It ends by partitioning the code into smaller pieces called *LTO partitions*. The partitioning happens with regard to the code being optimized, for example to minimize cross-partition edges.

LTRANS stage then performs optimizations decided by WPA stage, followed by local optimizations and code generation.

1.5 Evaluating GCC performance

As we previously noted, we focus on the GCC suite. In this section, we cover the experimental setup, evaluate compile time in selected programs and extract practical information on resources needed and anticipated use. The source code used to take these measurements is available online (see Attachments), allowing others to reproduce results presented and use it in future work.

1.5.1 Compiler

For further measurements, we the 5.3 release of GCC. This release is relatively fresh, supports most of the latest features, but formed a stable base for development and testing.

1.5.2 Tested software packages

Several opensource programs are available for testing, however a good candidate has to be selected in order to make testing straightforward and reproducible. We will choose testing applications based on the following criteria:

- written in C++ (preferred) or C,
- good compatibility with current GCC versions (5.x and 6.x),
- flexible and robust build system,
- mid to large codebase, and,
- preferably large monolithic binary.

It is surprisingly difficult to find projects that fit all of those requirements. For example the build systems of MySQL and Inkscape is very inflexible and has trouble with LTO compilation. GIMP consists of many plugins which does not pose any challenge for the current link-time optimizer.

The following applications were used in this work.

Firefox is already an established benchmark for GCC [GH10] and though earlier versiond were difficult to build with LTO, recent versions are polished and fullfill all the requirements. Is is the largest test case and though it takes only a few minutes to compile in standard setup, the time can be raised to many hours with certain optimizations enabled. We will discuss this in more detail later in Section 1.5.4. Besides the size, the codebase is also divers and makes use of modern C++ constructs. The specific version used in testing is *Firefox 48*.

Merkaartor is an OpenStreetMap editor written in C++ with medium sized code-base. It utilizes Qt framework, a lot of C++ constructs and links plenty of objects into a single binary. It also uses a lot of C++ constructs. The specific version used in testing is *Merkaartor 0.18.3-rc1*

SQLite is a SQL database engine in a single source file with a medium (to small) sized code-base. It is the simplest of all three and does not offer much challenge for the optimizer, but offers very quick turn time as well as an easy entry point for testcase minimization. The specific version used in testing is *SQLite 3.8.7.4*.

1.5.3 Experimental setup

Where relevant, the following machine was used for testing:

- Intel Xeon E3-1231-v3 @ 3.40GHz (Haswell),
- 32GB DDR3 RAM @ 1600MHz, 4 modules KHX1600C10D3/8G,

- 120GB Intel 520 SSD, SSDSC2CW120A3.

This setup is on the high-end of desktop computing, and much more than should be required for regular development.

The system was running 64bit Linux kernel 4.5 and standard Gentoo Linux installation. Memory and CPU usage measurements were taken using Linux Control Groups for whole compilation process, including GNU make and other tools. The data were sampled at one second intervals, which is more than enough. Total CPU usage is known precisely, as control groups keep cumulative counter. The activity at a given point is used only as a pointer as to how many cores are currently computing. The measurement is precise enough for memory too, as we are not allocating and freeing memory rapidly. In fact, most of memory allocation is done at the beginning of an analysis.

1.5.4 Compiling Firefox with LTO and IPA PTA

Firefox, in a standard configuration, can be compiled within minutes. See Figure 1.5 for a graph of the whole compilation (not only the `libxul.so`). The dip in CPU cores used around 16th minute is the linker for `libxul.so` being invoked, the following spike is a parallel compilation of a few unit tests and miscellaneous parts.

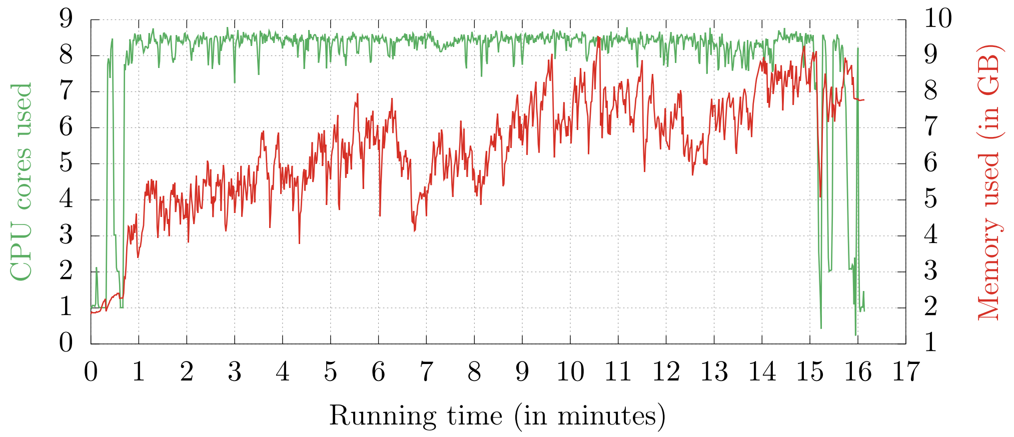
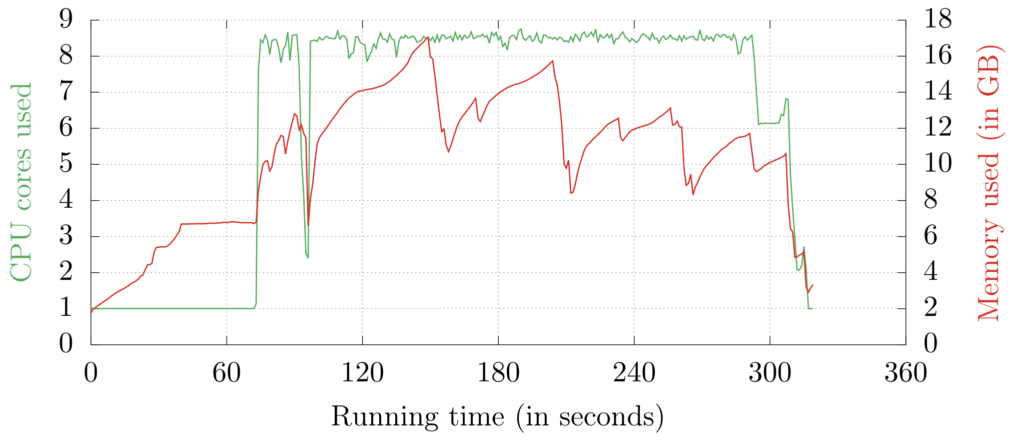


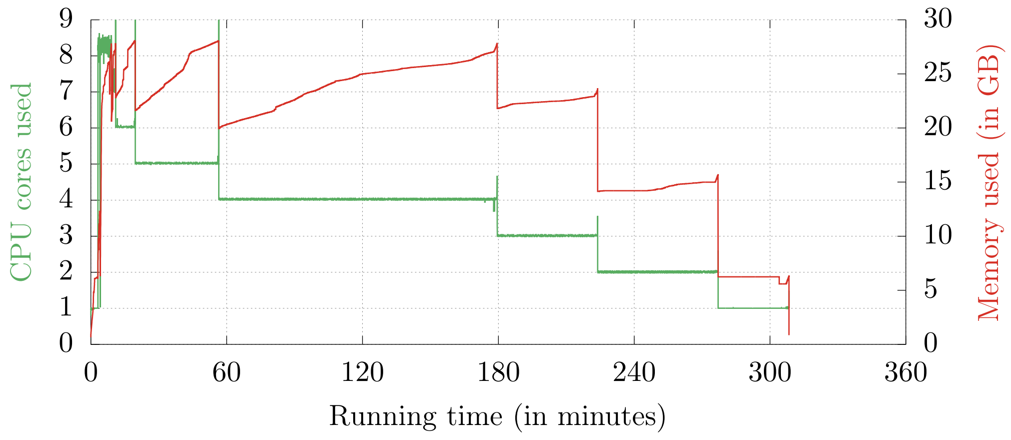
Figure 1.5: Compiling Firefox without LTO

Enabling LTO results in slightly longer compile time, but most of the time is spent during the LTO phase of `libxul.so`, which is plotted separately in Figure 1.6a. The link time is being actively improved by the GCC developers and substantial improvements are made between releases.

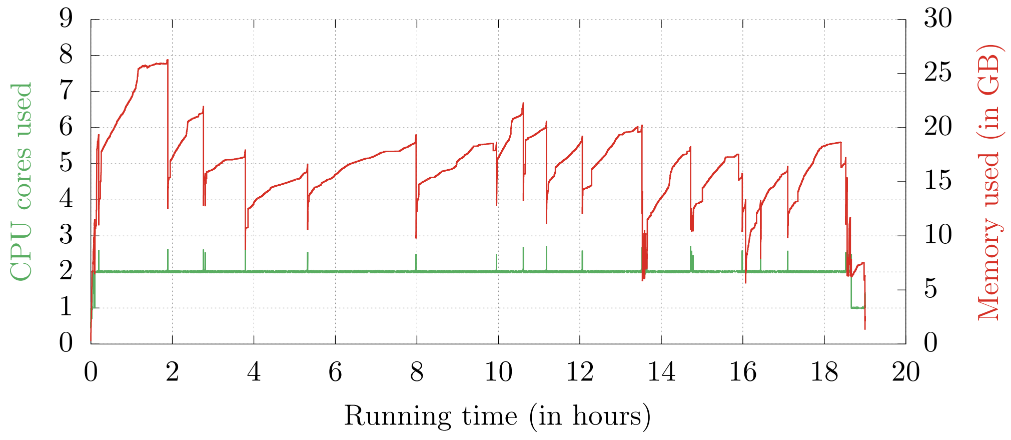
Enabling the interprocedural alias analysis pass causes the compilation to end abruptly due to insufficient RAM. See Figure 1.6b for details: each drop in utilized CPU cores shows a moment where one LTO process was killed by the kernel on out-of-memory condition (or finished successfully in later phases). This issue is remedied by running only 2 processes concurrently (see Figure 1.6c). However even in this setup the compiler used more than 27 GB RAM and it took around 18 hours to complete.



(a) `libxul.so` with `-flto=8`



(b) `libxul.so` with `-fipa-pta -flto=8`



(c) `libxul.so` with `-fipa-pta -flto=2`

Figure 1.6: Compiling `libxul.so` with different optimization options

1.5.5 Profiling GCC

To see what areas of GCC are worth improving, we used the `perf` profiler to record usage statistics for various GCC functions. Figure 1.1 shows top 20 used functions during LTO phase. It is not surprising to see a lot of `bitmap_*` functions, as a lot of passes use bitmaps for computations and storing results. Typical users are data-flow analyzers [Muc97] used many times through the compilation process, register allocator and points-to analysis. Enabling interprocedural points-to analysis (`-fipa-pta`) shows drastically different results, with functions `bitmap_ior_into` and `bitmap_elt_ior` taking almost all the CPU time. This clearly shows the pass could benefit from bitmap and/or algorithm optimizations.

Overhead	Cmd/Object	Symbol
2.92%	`ltrans/lto1`	`bitmap_set_bit`
2.19%	`ltrans/libc`	`_int_malloc`
1.92%	`ltrans/lto1`	`bitmap_clear_bit`
1.40%	`ltrans/lto1`	`ggc_internal_alloc`
1.26%	`ltrans/lto1`	`record_reg_classes`
1.16%	`ltrans/lto1`	`process_bb_lives`
1.07%	`ltrans/lto1`	`df_note_compute`
0.91%	`ltrans/lto1`	`bitmap_bit_p`
0.91%	`ltrans/lto1`	`df_ref_create_structure`
0.88%	`wpa   /lto1`	`inflate_fast`
0.87%	`ltrans/libc`	`_int_free`
0.77%	`ltrans/lto1`	`df_worklist_dataflow`
0.75%	`ltrans/lto1`	`inverted_post_order_compute`
0.71%	`ltrans/lto1`	`cselib_invalidate_regno`
0.70%	`ltrans/lto1`	`df_ref_record`
0.68%	`ltrans/lto1`	`pool_alloc`
0.61%	`ltrans/lto1`	`reload_cse_simplify_operands`
0.59%	`ltrans/lto1`	`bitmap_ior_into`
0.58%	`ltrans/lto1`	`cse_insn`
0.58%	`ltrans/lto1`	`df_reorganize_refs_by_reg`

Table 1.1: `perf` profile with `-flto`

Overhead	Cmd/Object	Symbol
80.99%	`ltrans      /lto1`	`bitmap_ior_into`
3.62%	`ltrans      /lto1`	`bitmap_set_bit`
2.11%	`ltrans      /lto1`	`find_what_var_points_to`
1.79%	`ltrans      /lto1`	`do_complex_constraint`
0.53%	`ltrans      /lto1`	`find`
0.40%	`ltrans      /lto1`	`bitmap_copy`
0.38%	`ltrans      /lto1`	`bitmap_bit_p`
0.36%	`ltrans      /lto1`	`bitmap_elt_insert_after`
0.35%	`ltrans      /lto1`	`add_graph_edge`
0.34%	`ltrans      /lto1`	`solve_constraints`
0.21%	`ltrans      /libc`	`_int_malloc`
0.18%	`ltrans      /lto1`	`bitmap_clear_bit`
0.13%	`ltrans      /lto1`	`topo_visit`
0.11%	`ltrans      /lto1`	`solution_set_expand`
0.10%	`ltrans      /lto1`	`record_reg_classes`
0.09%	`ltrans      /lto1`	`ggc_internal_alloc`
0.09%	`ltrans      /lto1`	`process_bb_lives`
0.09%	`ltrans/[unknown]`	`0x000000000061c7aa`
0.09%	`ltrans      /lto1`	`df_note_compute`
0.07%	`ltrans      /libc`	`_int_free`

Table 1.2: `perf` profile with `-flto -fipa-pta`

2. Alias Analysis

The goal of alias analysis is to enable optimizations across memory operations. It is used by other compiler components to disambiguate accesses to memory locations, enabling many optimizations. We discuss most commonly used decision methods, with focus on points-to analysis.

2.1 Common alias analysis methods

In the C language, a memory location is usually accessed by its name or via a pointer. Disambiguating two accesses is necessary for many optimizations, but missing an alias might result in incorrect code generated. See example in Figure 2.1. The second assignment to `b` might seem redundant, as `a` could not have changed. However, it is true only if the call to `some_fn` did not change variable `b`.

```
void set_call_set(void) {
  int a,b;
  [...]
  b = a + 1;
  some_fn(a, &b);
  b = a + 1;
  [...]
}
```

Figure 2.1: Example of the importance of alias information

Accurately disambiguating memory references may be arbitrarily complex. Many optimizations will however be possible even with a minimal aliasing information. Consider example in Figure 2.2. The loop seems to write zeroes into an array `a`. This is true if the pointer dereference can not change the pointer itself. Fortunately we do not need to examine any code not shown in the example. The C standard prohibits the dereference of `float*` to modify `float*` itself [ISO11]. This method is called *Type-Based Alias Analysis* (TBAA).

```
void fill_floats(void) {
  float* a;
  [...]
  for (int i = 0; i < 10; i++)
    *(a++) = 0;
}
```

Figure 2.2: Example of the importance of alias information

Alias analysis can not be solved accurately. We distinguish between may-alias and must-alias information, enabling us to give conservative answers. May-alias information indicates that the same memory can be accessed on at least one path in the control flow graph. On the other hand, must-alias information requires alias in all possible paths. Consider example in Figure 2.3. The information that “`p` points-to `x` or ’`y`’ is an example of may-alias, as it depends on the condition taken. The information “`q` points-to `x`” is an example of must-alias, as it holds on all paths in the example. Notice that must-alias always returns a single element, may-alias usually returns larger points-to sets.

```
void fill_floats(void) {
  int *p,*q;
  int x,y;
  [...]
  q = &x;
  if (x > y) {
    p = &x;
    [...]
  } else {
    p = &y;
    [...]
  }
  *p = 0;
  [...]
}
```

Figure 2.3: Example of may and must-aliasing

Alias analysis can be seen as a separate module of a compiler, which is accessed by optimization passes by the means of *alias oracle*. This usually is a function that given two memory accesses in a program answers if they can access the same memory location. The answer can be *yes*, *no* or *maybe*. A single oracle can apply multiple algorithms to determine the answer. The following three oracles are most often used:

Type Based Alias Analysis (TBAA) infers aliasing information from types and language-specific rules. For the C language [ISO11], an example of this mechanism has been shown in Figure 2.2 and discussed earlier. This method is very fast, as it only needs to inspect the types in question. For this reason, it is usually asked first and is able to distinguish many cases by itself.

Base and offset analysis is used especially for structures or arrays, where the access is composed of base pointer and an offset. The offset information might not be complete, but sometimes the range for offset is known. If the bases are distinct memory locations, the accesses do not alias. If the bases are provably the same memory locations, the offsets can be compared to see if the accesses can alias.

For example, consider the code in Figure 2.4. The base and offset would be able to decide that `a[5]` and `a[6]` do not alias, as their base is identical and offset differs by at least size of array type, `char`. On the other hand, it is unable to answer if `*p` and `a[0]` alias: though the base is provably the same, it is unknown what the offset is for `p`.

```
void base_offset(void) {
  char a[] = {..., 0};
  char *p = a;
  while ((*p) != 0 || (*p) != 'a')
    p++;
  a[5] = a[6];
  (*p) = a[0];
  [...]
}
```

Figure 2.4: Example of base an offset analysis

Points-to analysis is used in a case a memory access cannot be disambiguated by any simpler rule. A *points-to set* for a variable (pointer) is a set of memory locations the variable can be used to access. For example, a simple non-pointer variable can only be used to access itself (access by name). A pointer variable can be used to access other memory locations of which the address was taken. To disambiguate two pointer-dereferences the corresponding points-to sets have to be compared and if their intersection is empty, it is safe to assume they do not alias. If their intersection is non-empty, or some of the sets could not be computed, we must assume they do intersect to preserve correctness.

Compared to type based and base and offset analysis, points-to analysis is a time-consuming process and will be a focus of this chapter.

2.2 Points-to analysis

Both type-based analysis and base and offset analysis run in practically constant time. On the other hand, points-to analysis requires nontrivial processing and does not necessarily scale and we discuss it further. We first distinguish between the variants of the problem, as the approach to solve them differs wildly.

A *flow-sensitive* algorithm computes the alias information with regard to control flow. In the example in Figure 2.3 it would notice the different branches of `if` and provide information that "`p` points to `x` in the `if` branch" and similarly for the `else` branch. A *flow-insensitive* algorithm computes alias information without any regard to control flow. In the same example it would just output "`p` may point-to `x` or `y`".

Context sensitivity is a similar problem to flow sensitivity but in intraprocedural case. While flow sensitivity relies on control flow graph inside a single function, context sensitivity is based on callgraph. The callstack, or some part of it, is usually considered as a context.

Let us formally define the various alias-analysis types. See [Muc97].

Definition. *Flow-insensitive may-alias information* is a binary relation on the variables $A_{\text{FinMay}} \subseteq \text{Var} \times \text{Var}$. A pair (x, y) is in the relation if x and y can refer to the same memory location, possibly at a different place in the program, or at a different time during execution. This relation is symmetric, but is not transitive.

Definition. *Flow-insensitive must-alias information* is a binary relation on the variables $A_{\text{FinMust}} \subseteq \text{Var} \times \text{Var}$. A pair (x, y) is in the relation if and only if x and y always refer to the same memory location during the program execution. This relation is symmetric, but also transitive.

The flow-sensitive case is a more complicated, and can be examined both as a relation or function.

Definition. *Flow-sensitive may-alias information* is a ternary relation on the variables and program locations $A_{\text{FseMay}} \subseteq \text{Var} \times \text{Var} \times \text{Loc}$. A triplet (x, y, p) is in the relation if x and y can refer to the same memory location at the point p in program execution.

Definition. *Flow-sensitive must-alias information* is a ternary relation on the variables and program locations $A_{\text{FseMust}} \subseteq \text{Var} \times \text{Var} \times \text{Loc}$. A triplet (x, y, p) is in the relation if and only if x and y always refer to the same memory location at the point p in program, regardless of what the memory location is.

A similar definition could be used for context sensitivity, adding call context to the relation as well, or encoding it in the location. The specifics depend on the definition of context, as there are multiple possiblites. A context could be just a call site, or a path in callgraph from the entry point, possibly only to a limited depth.

2.2.1 Problem complexity

It is useful to know how difficult the problem of points-to analysis is. In this section we will review previous results showing the theoretical bounds for different problem variants.

The earliest classification is from Landi [LR91], who proves that computing flow-sensitive may- and must-alias information in the presence of single level pointers can be done in polynomial time. By adding more levels of indirection, as is common in most languages, the problem becomes NP-hard.

Later Horwitz [Hor97] proved that precise flow-insensitive alias analysis is NP-hard with only scalar variables and no heap allocations, though the result assumes unrestricted pointer dereference.

Chakaravarthy [Cha03] proved that when heap allocations are allowed the problem becomes undecidable, even if all the variables are scalar. The same articles also proves that the flow-insensitive variant is in P, if the variables are further restricted to well-defined types[1]. Although this is not always the case, it gives us hope that a successful alias analysis could be performed on a well-formed program.

[1]Known type and number of indirections

In practical applications, computing high-quality points-to analysis on a single function is achievable, but for interprocedural scope even the flow- and context-insensitive poses a considerable challenge.

2.2.2 Known algorithms and approaches

During the years, only a few algorithms have been developed and because alias analysis is a typical dataflow problem, there is little reason to expect a practical but fundamentally different algorithm.

Andersen's algorithm

First published by Lars Ole Andersen [And94], it is an *inclusion-based* algorithm is based on direct mathematical representation of aliases as points-to sets. That is, a points-to set for a given pointer p is a set S_p containing all locations pointer p can point to. Further expressions are then translated into set inequalities:

$$p_i = \&a \quad \rightarrow \quad a \in p_i \tag{2.1}$$

$$p_i = p_j \quad \rightarrow \quad p_j \subseteq p_i \tag{2.2}$$

$$p_i = *p_j \quad \rightarrow \quad \forall p_k \in p_j : p_k \subseteq p_i \tag{2.3}$$

The structure of proposed Andersen's flow-insensitive algorithm is shown in Figure 2.5.

1. Initialize variables using inequality 2.1.
2. Build a propagation graph using inequalities 2.2 and 2.3 with variables as vertices, propagations along edges.
3. Find strongly connected components in the grah and merge them into a single node.
4. Mark every node as changed.
5. For every changed node, reset its changed status, propagate the change along edges and mark nodes as changed if they were modified.
6. Repeat step 5. until no node is marked as changed.

Figure 2.5: Andersen's algorithm

Steensgaard's algorithm

The main problem of Andersen's approach is scalability. Elegant approach was developed by Bjarne Steensgaard [Ste96]. It is similar to Andersen's, but replaces inclusion-based constraints by equality-based constraints. Solving is then simplified to points-to class unification. This is why it is sometimes called *unification-based* algorithm. The unification can be done in almost linear time, which leads to a very fast and scalable algorithm, though sacrificing some precision.

The unification-based algorithms are less used, as the method is believed to be patented by Microsoft[Ste01]. It is being used in Open64 and was implemented in LLVM, but later removed in 2006 [Lat06] due to patent concerns. We expect this to change, as the patent has just expired while writing this thesis, in September 2016.

2.2.3 Further improvements

Steensgaard's algorithm can use Union-Find data structure for the unification, which is already extremely efficient [Tar75]. Andersen's algorithm has to deal with sets, and the choice of data structure for set management is harder. Two major improvements have been proposed to date, though none of them have been implemented in a production compiler.

Bloom filters

The use of Bloom filters was first proposed by Nasre et. al [Nas+09]. They are very space efficient and perform well on certain operations, as is query and union. Some implementations can also perform interesection, but with decresed precision. The complete lack of the ability to enumerate elements was worked around by introducting multiple dimensions for multi-level pointers. In this scheme, a pointer could be easily dereferenced upto a constant depth and after that, the algorithm answers conservatively.

We will revisit the use of Bloom filters in later chapters.

Binary decision diagrams based algorithms

A Binary decision diagram (BDD) is a data structure used to represent boolean functions. It can be easily extended to represet relations by encoding characteristic function of given relation, and the complete alias information as well. Multiple algorithms based on the BDDs were developed [WL04; Bie05], but most of them lack public and usable code for further development. The major issue with the use of BDDs is that they heavily rely on the correct variable ordering. Choosing wrong ordering quickly results in size explosion and speed decrease. However the BDD approach seems promising for loss-less representations.

2.3 Current state in compilers

There are not many modern compilers with open code that can be examined and improved upon. One of the players is GCC, that has been around since 1985 (1.x release was in 1991) and is the most widely used open source compiler today. The younger competitor is LLVM/Clang, first released in 2003. It is written in C++, is supported by Apple since 2005, and due to its age has more modern design, and is generally deemed to be easier to extend and work with. Other competitor is Open64, which lacks community support, but is still being developed by some groups.

Many researchers also focus on Java compiles and algorithms, and though many techniques can be used for C and C++, Java is very different language, in that it has just in time compiler (JIT), and does not have pointers in the classic sense, only references, which simplifies some cases.

There are many more compilers available, but most of them are proprietary or not maintained, as for example the Intel C++ Compiler (ICC), VisualC++, SUIF and IMPACT.

It is very hard to compare many of the published results, as the implementations are not public, and mostly implemented for compilers that are unable to keep up with current C/C++ standards and successfully build modern (and big) projects. Many of the results are computed outside the compiler and never tested. Even if they were, there is no simple metric that could be used for comparison. The results rely on previous optimization passes, constraint generator, chosen granularity (wether to consider structure members or arrays) and finally queries asked by the compiler in later optimization phases.

In the rest of this chapter we briefly summarize the state of art of alias analyzers in open-source compilers.

GCC

GCC has a good support for TBAA and Base-offset analysis, intraprocedural points-to analysis, but lacks a good interprocedural points-to analysis. We discuss details in Chapter 4.

LLVM/Clang

As LLVM is very modular, it contains multiple alias analysis passes. In the core package there is `-basic-aa` pass, providing local alias information using many language-specific facts. It is similar to GCC's TBAA and Base-offset analysis.

Additionally, the `poolalloc` package provides a `-globalsmodref-aa` pass, providing context-sensitive alias information for global variables similar to GCC's `ipa-reference pass`. It also implements the Steensgaard's algorithm in the `-steens-aa` pass, and Andersen-style context and field-sensitive points-to analysis in `-ds-aa`.

Open64

Open64 traditionally implements TBAA, Base and offset analysis and points-to analysis using Steensgaard's algorithm. A new context-sensitive andersen-style alias analysis has been implemented in 2013-2014 [Sui+14], though the context sensitivity seems to be only partially implemented.

3. From Bloom filters to Bloomaps

During points-to analysis the datastructure is almost as important as the algorithm used. In the case of GCC, the structure chosen is a hybrid of bitmap and a linked list. This works fine for small and dense data, but not so well with large data. Converting to a better data structure is relatively easy task, but what data structures are available? We start by examining the needs of a typical Andersen-style algorithm and comparing theoretical complexities of various well known data structures. In the rest of this chapter we will describe a new enhancement of Bloom filters called Bloomaps, tailored specifically for the use in points-to analysis.

3.1 Requirements

As a basic data structure for points-to analysis, we need a data structure holding sets of integers that is compact and has the following operations.

- `INSERT(set, element)` – inserts `element` into `set` and returns if the structure has been changed by the insertion.
- `QUERY(set, element)` – checks if `element` is part of `set`.
- `INTERSECTION_EMTPY(set1, set2)` – checks if intersection of `set1` and `set2` is empty.
- `ENUMERATE(set)` – lists all elements in a `set`.
- `UNION(set1, set2)` – merge `set2` into `set1`.

The algorithm uses the following operations in the following way:

- Initialization: `INSERT` used to populate points-to sets with memory locations assigned to them.
- Propagation: `UNION` is used for every copy constraint, `ENUMERATE` and `UNION` for dereferences. Majority of time and memory is consumed by the propagation stage.
- Oracle queries: `QUERY` is used for set membership, `INTERSECTION_EMPTY` for set disjointness

In other words, the basic operations can be slower, `INTERSECTION_EMPTY` and `UNION` has to be fast. There are a few other considerations:

- `UNION` will be called on the same pairs over and over again, the difference will usually be in just a few elements.
- The average number of stored elements will be small, and the data sparse.

- Some sets may grow very large, containing almost every element possible.
- Low memory overhead is required, as the number of sets is in the order of number of elements inserted.

Traditionally GCC uses two basic types to represent sets. One called `sbitmap`, which is plain bit array of a given size, and second `bitmap`, which is designed for storing sparse bitmaps. The later consists of a linked list of blocks, each block containing 128 bits, and allows to skip long sequences of empty blocks. The implementation is relatively memory efficient and performs well on many dataflow solvers including bitwise `AND` and `OR`. In case only a few bits are set in a block, the worst-case complexity of $\mathcal{O}(n)$ for most operations comes into play and the stucture becomes unusable in points-to solver. This is expected to happen relatively often. For example, Firefox has tens of millions of declarations, each of them can find itself in a set, with high probability to be a single bit in that set. A program can be made to generate arbitrary points-to sets. For this reason it is unrealistic to expect points-to sets to have some structural properties that could lead us to efficient and precise data structure. This leads us to consider non-exact data structures.

3.2 Bloom filters

A Bloom filter is a classical probabilistic data structure, invented by Burton Howard Bloom [Blo70]. The goal is to provide a data structure that has some nonzero probability of *false-positive*, but zero probability of *false-negative.* This is accomplished by taking a bit field of m bits, k hash functions, and hashing every element into k different bits, writing 1 on insertion, and checking if every position contains 1 on query.

The Bloom filter has immediate applications in some areas, for example caching: it is a good idea to ask a filter if an element is in the cache. If the answer is no, we need to get it elsewhere. If the answer is yes, we can look into the cache, and in the worst case it is not there (an ocurrence false-positive).

Basic operations can be implemented with the following time complexity (provided the hashes can be computed in constant time, which is often possible):

- `QUERY` in $\mathcal{O}(1)$ time.
- `INSERT` in $\mathcal{O}(1)$ time.
- `UNION` in $\mathcal{O}(m)$ time (bitwise OR).
- `BITWISE_INTERSECTION` in $\mathcal{O}(m)$ time.
- `DELETE, ENUMERATE, RESIZE` not supported[1].

Unlike in simple bitmaps, bitwise intersection of Bloom filters is not equal to set intersection. Denote by $BF(A)$ a Bloom filter created from empty filter by

[1]Though there are variations that support these oprations, only `RESIZE` is usually possible without drastic changes to the structure.

inserting elements from A one by one. Then for A, B it does not hold that $BF(A \cap B) = BF(A) \cap BF(B)$. While the inequality $BF(A \cap B) \subseteq BF(A) \cap BF(B)$ holds, it is nowhere near the equality. Most imporantly `INTERSECTION_EMPTY` is hardly reasonably accurate, because a single bit in `BITWISE_INTERSECTION` causes the filter to be non-empty.

3.3 Bloom filter intersection

Although a Bloom filter intersection is easily computed with bitwise `AND`, it is rarely accurate. As proven in [Bos+08], the probability that $BF(A \cap B) = BF(A) \cap BF(B)$ is:

$$p = (1 - 1/m)^{k^2 \cdot |A - A \cap B| \cdot |B - A \cap B|}.$$

For `INTERSECTION_EMPTY`, we can further simplify the formula by considering two cases: $|A \cap B| > 0$ and $|A \cap B| = 0$. Assuming that $|A \cap B| = 0$, the probability that $BF(A) \cap BF(B)$ is also empty. It is as follows:

$$p_{empty} = (1 - 1/m)^{k^2 \cdot |A| \cdot |B|}.$$

Jeffrey and Steffan [JS11] showed a slightly improved bound with partitioned Bloom filters.

$$p_{empty} = \left(1 - \left(1 - \frac{k}{m}\right)^{|A| \cdot |B|}\right)^{k}.$$

Partitioned Bloom filter has a separate partition for each hash function of size m/k, therefore it is enough to have one empty partition to consider the filter empty, as every query would result in false in this empty partition.

The same paper also proved that pure Bloom filter intersection is more memory consuming than storing inserted elements in an linked list. In the next section, a hybrid solution is provided that can be used with both of these approaches, based on the time requirements.

3.4 Bloom filter enumeration

It is immediately clear that vanilla Bloom filter does not support element enumeration. For example the simplest filter holding 1 elements and answering with false-positive probability 0.5 would have to enumerate half the universe U, which may as well be impossible for $U = \mathbb{N}$.

Let us briefly summarize a few options most commonly used:

Enumerate entire universe. Without any extra information on what may be contained in a filter, we have to check for every element in the universe. This is possible for small and dense universe, and appropriately sized filter. Even for almost empty filter, this approach takes $\mathcal{O}(|U|)$ time.

Keep a Queue-of-queries for each filter. This approach is suggested by [JS11]. This idiom is to keep a queue of elements inserted into the filter, which is evaluated in case the elements need to be enumerated. The queue usually holds each element as many times as it has been inserted. The filter has limited capacity, so as long as we do not insert elements many times over excessively, the list can not grow indefinitely. However, this approach does not work well with `UNION`, as the lists would have to be either concatenated (in which case the lists would grow exponentially) or pruned after each union, which would be essentially same as using bitmaps.

Alternative structures. A structure has been proposed by Michael Goodrich [GM11], called Invertible Bloom Lookup Tables (IBLT). The problem of IBLT is that they have non-zero probability of being unable to produce a complete list of entries, and do not provide the advantages of classic Bloom filters, as a fast intersection and membership queries. This said, we will not attempt to use them, although they are an interesting structure for future work and may find its use in other optimizers.

Neither of these methods are good for use in points-to analysis, as our universe can be large (tens of millions of lines of code), and the number of filters is about the same size as the universe. We introduce a new approach, a compromise between the two above.

Keep a single Queue-of-queries for all filters. The number of sets in points-to analysis prohibits the use of queue-of-queries for each filter. The universe enumeration could be used, but is wasteful as only some variables will ever be inserted in the set. However, we can keep a global queue-of-queries for all filters used in the algorithm. Each element inserted will be recorded in an indexed data structure, and parts of this structure will be searched and evaluated later, when enumeration is requested.

Choosing a good data structure is still important, as we still need low overhead and preferably fast insertion. Assuming our universe is all 32 bit integers, a single bit array of every item ever inserted in under 512 MB. This is reasonable, but storing even 8 bits for each element would result in 4 GB, which is not. We show later in this chapter an efficient indexed bit array with insertion in $\mathcal{O}(1)$.

3.5 Bloomaps and Families

Bloomap Family with parameters (m, k, s) is a datastructure that maintains a list of Bloomaps of the same parameters, and indexed representation of used parts of the universe in union of all its Bloomaps, capable of enumeration.

A *Bloomap* is an enhanced Bloom filter, belonging to a single family, capable of executing `INSERT` and `QUERY` itself, and `ENUMERATE`, `UNION` and `INTERSECTION` within its family.

Bloomap with parameters (m, k, s) is constructed from a partitioned Bloom filter with k partitions, each for one hash function, with the addition of a *side index* containing s bits. The side index is used as another partition in the Bloom filter, however with simpler hash function that is easily inverted (for example a simple `SHIFT` and `AND` with a mask).

Furthermore, the Bloomaps and their families need to fulfill these conditions:

- When a new item is inserted into a Bloomap, it is also inserted into the family.
- A Family has to enumerate all items inserted into it's Boomaps for any given hash.

Figure 3.4 shows Bloomap and BloomapFamily prototypes. Here `hash_set` is some data structure capable of storing a set of elements from universe associated with a given hash. In C++, `hash_map<vector<universe_type>>` could be used as a naive data structure, but we offer a better solution later in this chapter. See Figure 3.5 for pseudocode implementation of `INSERT` and `ENUMERATE` functions. Overview of Bloomap structure is shown in Figure 3.1.

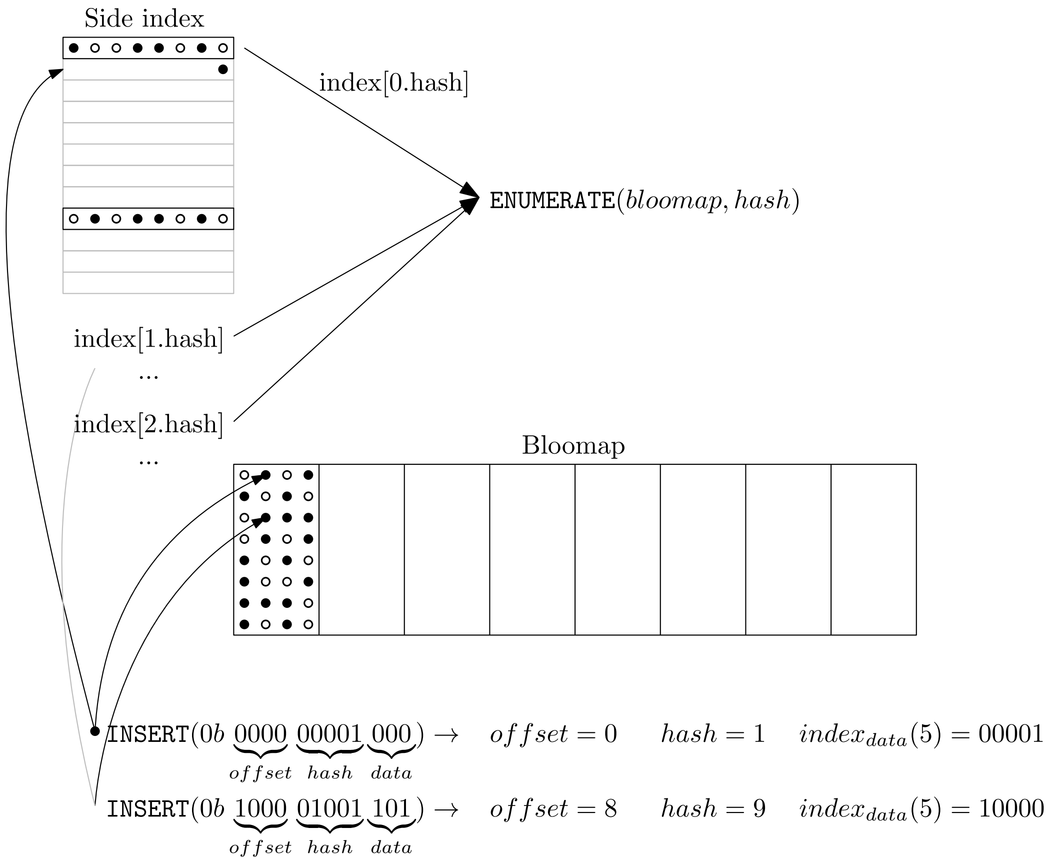

Side index
index[0.hash]
ENUMERATE(bloomap, hash)
index[1.hash]
...
index[2.hash]
...
Bloomap
INSERT(0b 0000 00001 000) → offset = 0 hash = 1 index_data(5) = 00001
offset hash data
INSERT(0b 1000 01001 101) → offset = 8 hash = 9 index_data(5) = 10000
offset hash data

Figure 3.1: Overview of Bloomap structure

Both stuctures are pretty straightforward, as `INSERT` is a regular function for Bloom filter insertion, with the added partition for side index and family universe insertion.

While insertion in Bloom filter is in $\mathcal{O}(1)$, inserting into a universe may be $\mathcal{O}(\log n)$ for tree-based implementations or amortized $\mathcal{O}(1)$ for hash-based implementations. There probably is not a better solution in generic case, but we suggest a worst-case $\mathcal{O}(1)$ for 32 bit integers (dense sequence of ids starting from zero).

The Bloomap achieves the following theoretical time complexity:

- `QUERY` in $\mathcal{O}(1)$ time.
- `INSERT` in $\mathcal{O}(1)$, assuming $\mathcal{O}(1)$ insertion into universe index.
- `ENUMERATE` in $\mathcal{O}(|U|)$ in worst case, assuming universe index suggested in section 3.5.1 and $|U|$ the size of the index.
- `UNION` in $\mathcal{O}(m)$ time.
- `BITWISE_INTERSECTION` in $\mathcal{O}(m)$ time.
- `DELETE`, `RESIZE` not supported.

Though the theoretical complexity of `ENUMERATE` seems very bad, leading potentially to tens of millions of queries. Here the constant is important, as the Bloomap will rarely be completely full and most of the queries will be filtered by the side index and universe index. We can also further optimize the algorithm to reject enumeration of Bloomap that has more elements than designed for.

3.5.1 Compact representation of dense integer universe

Representing universe requires storing sets for different hashes. It is wasteful to store them in a linked-list, trees or even hash tables, as a humble bit array fulfills the task. A little unusual form of a bit array has been used, in order to achieve less allocations and good space efficiency.

As mentioned above, we will split the value to `offset`, `hash` and `data` at binary boundaries. This means we can simply concatenate the values to get the represented integer. We can now organize the data into *buckets* and *superbuckets* in the following way:

- Each `offset` has it's own *superbucket*.
- Each *superbucket* contains a *bucket* for every `hash` value.
- Each *bucket* contains a bit for every `data` value.

`element = offset·hash·data` ⇔ `universe[offset·hash].bit[data]`

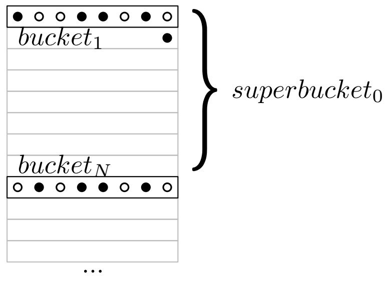


Figure 3.2: Bucket and superbuckets in an array.

The structure is illustrated in Figure 3.2 and pseudocode implementation in Figure 3.3. Memory allocation is expected to be done automatically in `vector` class and the array should be resized on first access beyond current boundary. This allows the structure to occupy only as much memory as is necessary to store a set of size at most $\mathcal{O}(\max(E))$, where E is the number of elements inserted. In this case, a bucket has 64 bits, so it will fit up to 6bit data. Number of bits in hash and offset is not important and should be chosen by the size of Bloomap's side index.

```
struct superbucket {
  uint64_t bits[];
};

struct universe_index {
  vector<struct superbucket> superbuckets;
};

void universe_index::INSERT(offset, hash, data) {
  superbuckets[offset]->bits[hash].bit[data] = 1;
}

vector<element> universe_index::ENUMERATE(hash) {
  vector<element> candidates;
  for (sb in superbuckets) {
    for (i = 0 .. 63) {
      if (sb->bits[hash].bit[i])
        candidates.append(bit);
    }
  }
  return candidates;
}
```

Figure 3.3: Bucket and superbuckets prototype and pseudocode

```
struct Bloomap {
  BloomapFamily f;              # Family this maps belongs to
  int m,k,s;                    # Filter parameters

  int index[s];                 # Side index
  int partitions[k][m/k];       # Regular filter partitions
};

struct BloomapFamily {
  vector Bloomap;               # Owned bloomaps
  int m,k,s;                    # Filter parameters

  hash_set universe;            # Indexed universe
}
```

Figure 3.4: Bloomap and BloomapFamily prototypes (in pseudocode)

```
void Bloomap::INSERT(element) {
  # Decompose element to offset, index_hash and data.
  (offset,index_hash,data) := element;
  # Insert into side index of a bloomap.
  index[index_hash] := true;
  for (i := 1..k) {
    partitions[i][hash_fn(i,element) % m/k] := 1
  }
  # Insert into universe index of a family
  f.universe[index_hash] += element;
}

bool Bloomap::QUERY(element) {
  for (i := 1..k) {
    if (partitions[i][hash_fn(i,element) % m/k] == 0)
      return false;
  }
  return true;
}

void Bloomap::UNION(another) {
  for (i := 1..s) {
    index[i] |= another.index[i];
  }
  for (i := 1..k, j := 0..m/k) {
    partitions[i][j] |= another.partitions[i][j];
  }
}

vector Bloomap::ENUMERATE() {
  vector list = ();
  # Check each element in side index
  for (i := 1..s | index[i] == true) {
    # Test for every element universe lists for this index
    for (element in f.universe[i] && this.QUERY(element))
      list += element;
  }
  return list;
}
```

Figure 3.5: Basic Bloomap operations (in pseudocode)

4. Using Bloomaps in points-to analysis

In this chapter we discuss the internals of points-to analysis as implemented in the GCC compiler and how it was augmented by the use of Bloomaps instead of regular bitmaps. Later we show how the change affected GCC and discuss future work.

4.1 Points-to analysis in GCC

The points-to analysis in GCC is implemented in two files. In `tree-ssa-alias.c` contains the alias oracle and the TBAA and Base-offset algorithms (see sections 2.1 and2.1). Function `refs_may_alias_p(tree,tree)` wraps most of the functionality, but a few others exist to check for aliasing with global memory, call clobbers and other special cases. It also contains interface to alias analysis on RTL objects, which query the oracles listed above.

The points-to information is stored in a structure `pt_solution`, which is computed by algorithm in `tree-ssa-structalias.c`. It is an implementation based on [PKH04; HT01], and is an Andersen-style algorithm. An overview of this algorithm is on Figure 4.1.

The implementation has two modes. One for intraprocedural points-to analysis (PTA) and second for interprocedural points-to analysis (IPA PTA). In our case this code reuse complicates the development. We want to keep the intraprocedural analysis as it actually performs well, and modify the interprodecural version which has performance issues as discusses in earlier chapters.

4.1.1 Improving the implementation

In this work, we have temporarily duplicated the code into `ik-structlias.c`, which was then modified to implement only the interprocedural case. This is just a temporary solution which enables us to run the original and modified points-to algorithm (called IPA KPTA) in a single execution and directly compare the results. Furthermore, while benchmarking only the points-to algorithms code changes, the rest of the compiler is identical. The most important reason is to avoid case separation in each function. The actual algorithm needs adjustments: we not only need to pass different data types, but some operations are no longer supported, and some operations should be used with greater care than they are now. After inspecting the code, it became clear that the two algorithms need to be separated.

To split the code, a few modifications has been done. Common functions were marked static and renamed to avoid confusion. New query functions were added to `tree-ssa-alias.c`, which ask both the original and the new points-to oracle if the data is available. A new pass has been created, called `kpta`, which is controlled by new command line options:

- `-fipa-kpta` is an analog to `-fipa-pta` and instructs the compiler to run

1. Allocate `varinfo_t` structure for each variable. `varinfo_t` includes metadata and a solution set.
2. Find constraints in the form of `p = &q` (direct constraints), and use them to initialize the solution sets.
3. Find constraints in the form of `p = q` (copy constraints), and use them to build a constraint graph on variables. For example for `p = q` constraint an edge `q` $\rightarrow$ `p` is inserted to the graph.
4. Find other constraints in form of `*p = q` or `p = *q` or containing field offsets (complex constraints) attach them to their corresponding vertices in the constraint graph.
5. Find and contract strongly connected components in the graph.
6. Put all graph vertices into a worklist.
7. Take a vertex from the worklist. Process all complex constraints (possibly adding more copy edges to the graph) and propagate the set along the copy edges, including complex constraints. Put all vertices modified by this operation into the worklist.
8. Repeat step 7. while there are elements in the worklist.

Figure 4.1: GCC implementation of Andersen's algorithm

the new IPA KPTA algorithm during LTO phase, just after the original IPA PTA pass (if enabled).

- `--param kpta-bloomap-size=n` forces a Bloomap of specific size.
- `--param kpta-bloomap-precision=p` forces a Bloomap of specific precision. The value passed is inverted precision in percent, so a value of 100 will result in a precision of 1%.

4.1.2 Integrating Bloomaps

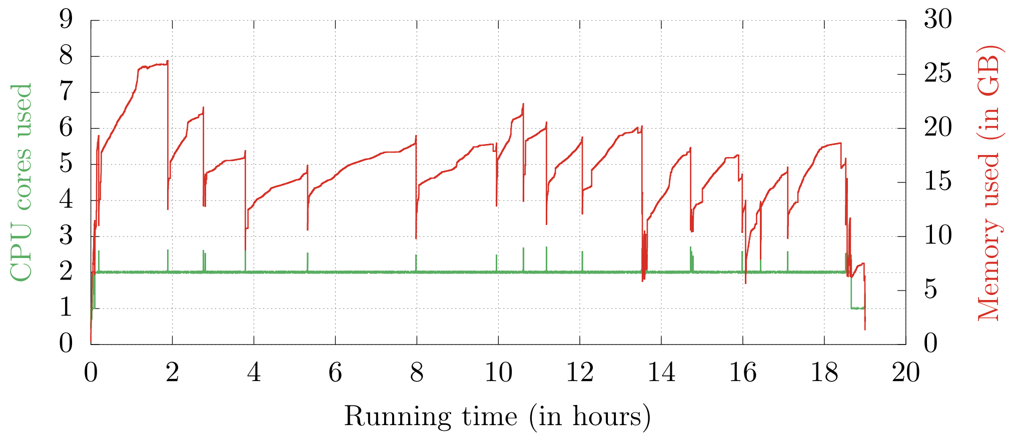
Integrating Bloomaps was relatively straightforward. The following steps were necessary, as some operations do not map well to Bloomap operations.

- The main loop in `solve_graph()` keeps two solution sets: a current one and one from previous iteration. When new elements are to be propagated (via complex constraints), only the difference is examined to make changes. This is a nice optimization for classical bitmaps. However, Bloomap has no easy method to list difference and would have to be enumerated.
- Due to historical reasons, two identifiers were used in the algorithm. This is no longer necessary and one of them was removed.

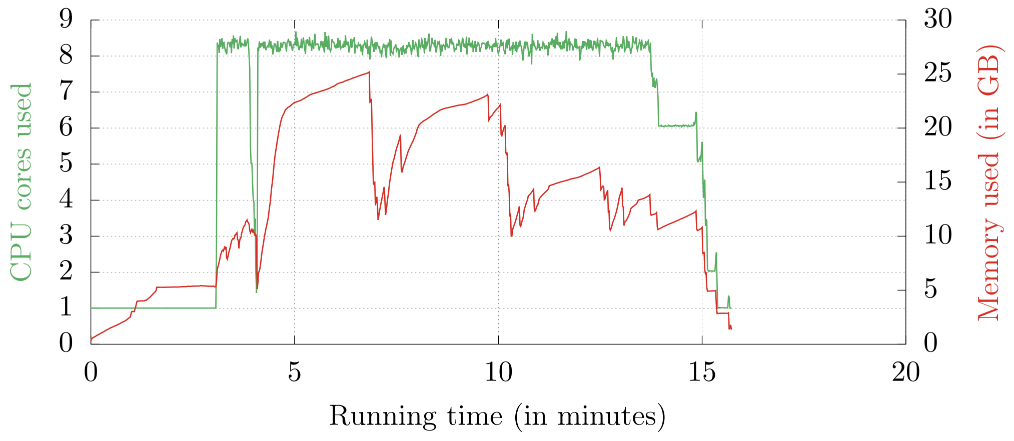
- Finished solutions were deduplicated, merging same sets into one. This results in a less memory use, but is not worth the effort for Bloomaps, as they are already very space efficient.
- Unnecessary bitmap enumerations were removed and the remaining ones optimized to reduce the number of passes.

4.2 Performance evaluation

By using Bloomaps instead of classical bitmaps, we have improved significantly both the compile time and memory usage. See Figure 4.2 for comparison. We measured compile time in hours for the old IPA PTA algorithm, and we could only utilize 2 cores due to memory requirements. The improved IPA KPTA algorithm finishes in just 16 minutes, utilizing approximately the same memory, but working in 8 threads instead of 2. Though the memory use is still not ideal, taking approximately extra 10 GB (see Figure 1.5 in the first chapter). It is now viable to enable IPA KPTA by default during LTO phase.



(a) `libxul.so` with `-fipa-pta -flto=2`



(b) `libxul.so` with `-fipa-kpta -flto=8`

Figure 4.2: Comparison of build time with old and improved algorithm

We have also analyzed the precision lost by using Bloomaps instead of an exact data structure. There is no good metric to use, as a single difference in answer might add or prevent some following queries. We implemented a special procedure that compares IPA PTA and KPTA results. For variables where boths points-to sets are computed, we provide a comparison by intersecting all pairs of sets. A percentage of empty intersections is computed for both the old and new algorithm. Any difference is result of the imprecision introduced, either directly caused by imprecise intersection, or by conservative propagation earlier.

This approach is not ideal, as the absolute percentage varies greatly between problem instances, but the relative difference is mostly stable. The precision achieved was always within 2% of exact datastructure and some minor improvements were made using additional checks via enumeration.

4.3 Future work

Our results demonstrate that Bloomaps are a competetive alternative to classical bitmaps. Not only will be interesting to use Bloomaps in other algorithms that do not require precision sets, but also further improving on IPA KPTA. It is possible to save even more memory by using deduplication or pruning of non-interestings sets (those that are too full or could point to anything). More precise analysis could be devised by starting with bitmaps and converting to Bloomaps in case they get too big to process, or by prefering bitmaps for sets that have to be enumerated, or are expected to be more important than others.

It is also possible to decide Bloomap size and/or precision at runtime, as the approximate number of pointers and dereferences is available beforehand. A further extension to the Bloomaps using ideas in [Guo+06], though it is not clear how the side index should be constructed.

The precision of the algorithm can also be improved. It handles some cases as function parameters and return values too conservatively. As the improved algorithm is now able to analyze most programs in existence, we hope other developers will join the effort and improve the constraint generation as well.

As the algorithm now scales, it should be possible to generate constraints during compilation, compute points-to sets during WPA phase (for whole program rather than partitions) and stream the results into LTRANS. This would result in even better propagation and is similarly implemented in Open64.

Conclusion

In this work we have identified bitmaps as one of the most used data structure in GCC and one of its biggest users, the interprocedural points-to analysis. We have enhanced this algorithm with a new data structure based on Bloom filters, the Bloomap. In the link-time optimization of Firefox, we decreased memory usage from 13 GB per process to 3 GB per process during link-time optimization phase, and build time has been decreased from 18 hours with the old pass to around 16 minutes with the improved pass. This is a major improvement and enables us to analyze programs that could not have been analyzed before without excessive resource use.

To our knowledge it is the first open-source implementation able to compute interprocedural points-to analysis for projects like Firefox using reasonable resources. It works well in production environment and has been checked to give conservatively correct results. The code currently exists as a patch to GCC and We work toward including the code in mainline GCC.

Bibliography

[AC76] F. E. Allen and J. Cocke. "A Program Data Flow Analysis Procedure". In: *Commun. ACM* 19.3 (Mar. 1976), pp. 137–.

[All75] F. E. Allen. "Interprocedural Analysis and the Information Derived by It". In: *Programming Methodology, 4th Informatik Symposium.* London, UK: Springer-Verlag, 1975, pp. 291–322.

[And94] L. O. Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. 1994.

[ASU86] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[Bie05] K. Bierhoff. "Alias analysis with bddbddb". In: *Star Project Report* (2005), pp. 17–754.

[Blo70] B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13 (1970), pp. 422–426.

[Bos+08] P. Bose et al. "On the false-positive rate of Bloom filters". In: *Information Processing Letters* 108.4 (2008), pp. 210–213.

[Bri+07] P. Briggs et al. "WHOPR-Fast and Scalable Whole Program Optimizations in GCC". In: *Initial Draft* (2007).

[Cha03] V. T. Chakaravarthy. "New Results on the Computability and Complexity of Points–to Analysis". In: *SIGPLAN Not.* 38.1 (Jan. 2003), pp. 115–125.

[GCCInt] GCC developers. *GCC Internals.* URL: `https://gcc.gnu.org/onlinedocs/gccint/` (visited on 12/19/2016).

[GH10] T. Glek and J. Hubička. "Optimizing real world applications with GCC Link Time Optimization". In: *CoRR* abs/1010.2196 (2010).

[GM11] M. T. Goodrich and M. Mitzenmacher. "Invertible Bloom Lookup Tables". In: *CoRR* abs/1101.2245 (2011).

[Guo+06] D. Guo et al. "The Dynamic Bloom Filters". In: *In Proc. IEEE Infocom.* 2006.

[HL07] B. Hardekopf and C. Lin. "The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 290–299.

[Hor97] S. Horwitz. "Precise Flow-insensitive May-alias Analysis is NP-hard". In: *ACM Trans. Program. Lang. Syst.* 19.1 (Jan. 1997), pp. 1–6.

[HT01] N. Heintze and O. Tardieu. "Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second". In: *SIGPLAN Not.* 36.5 (May 2001), pp. 254–263.

[ISO11] I. O. for Standards. *ISO/IEC 9899:2011 – Information technology – Programming languages – C.* 2011.

[JS11] M. C. Jeffrey and J. G. Steffan. “Understanding Bloom Filter Intersection for Lazy Address-set Disambiguation”. In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA ’11. San Jose, California, USA: ACM, 2011, pp. 345–354.

[KM06] A. Kirsch and M. Mitzenmacher. “Less hashing, same performance: Building a better Bloom filter”. In: *European Symposium on Algorithms.* Springer. 2006, pp. 456–467.

[Lat02] C. A. Lattner. “LLVM: An infrastructure for multi-stage optimization”. PhD thesis. University of Illinois at Urbana-Champaign, 2002.

[Lat06] C. A. Lattner. *Removing DSA from LLVM.* 2006. URL: `http://lists.llvm.org/pipermail/llvm-dev/2006-December/007557.html` (visited on 07/12/2016).

[LR91] W. Landi and B. G. Ryder. “Pointer-induced Aliasing: A Problem Classification”. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’91. Orlando, Florida, USA: ACM, 1991, pp. 93–103.

[Mor98] R. Morgan. *Building an optimizing compiler.* Digital Press, 1998.

[Muc97] S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[Nas+09] R. Nasre et al. “Scalable context-sensitive points-to analysis using multi-dimensional bloom filters”. In: *Asian Symposium on Programming Languages and Systems.* Springer. 2009, pp. 47–62.

[OpenHub] Black Duck Software, Inc. *Open Hub.* URL: `https://www.openhub.net/` (visited on 12/29/2016).

[PKH04] D. J. Pearce, P. H. J. Kelly, and C. Hankin. “Efficient Field-sensitive Pointer Analysis for C”. In: *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* PASTE ’04. Washington DC, USA: ACM, 2004, pp. 37–42.

[PPR05] A. Pagh, R. Pagh, and S. S. Rao. “An optimal Bloom filter replacement”. In: *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics. 2005, pp. 823–829.

[SS00] M. Streckenbach and G. Snelting. *Points-To for Java: A General Framework and an Empirical Comparison.* Tech. rep. 2000.

[Ste01] B. Steensgaard. “US patent 6202202, Pointer analysis by type inference for programs with structured memory objects and potentially inconsistent memory object accesses”. Pat. 6202202. 2001.

[Ste96] B. Steensgaard. “Points-to Analysis in Almost Linear Time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 32–41.

[Sui+14] Y. Sui et al. “Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation”. In: *Software: Practice and Experience* 44.12 (2014), pp. 1485–1510.

[Tar75] R. E. Tarjan. “Efficiency of a Good But Not Linear Set Union Algorithm”. In: *J. ACM* 22.2 (Apr. 1975), pp. 215–225.

[WL04] J. Whaley and M. S. Lam. “Cloning-based context-sensitive pointer alias analysis using binary decision diagrams”. In: *ACM SIGPLAN Notices*. Vol. 39. 6. ACM. 2004, pp. 131–144.

[ZR08] X. Zheng and R. Rugina. “Demand-driven alias analysis for C”. In: *ACM SIGPLAN Notices* 43.1 (2008), pp. 197–208.

[GCC05] GCC developers. *Link-Time Optimization in GCC: Requirements and High-Level Design.* 2005. URL: `https://gcc.gnu.org/projects/lto/lto.pdf` (visited on 12/29/2016).

[LLV] LLVM developers. *LLVM Alias Analysis Infrastructure.* URL: `http://llvm.org/docs/AliasAnalysis.html` (visited on 07/12/2016).

List of Figures

List of Tables

List of Abbreviations

BDD	Binary Decision Diagram
CLA	Compile Link Analyze
DSA	Data Structure Analysis
GCC	GNU Compiler Collection
GNU	GNU is Not Unix
GNU GPL	GNU General Public License
IBLT	Invertible Bloom Lookup Table
ICC	Intel C++ Compiler
IL	Intermediary Language
IPA	InterProcedural Analysis
JIT	Just In Time
LLVM	Low Level Virtual Machine
LTO	Link-Time Optimization
LTRANS	Local TRANSformation
PTA	Points-To Analysis
RTL	Register Transfer Language
TBAA	Type Based Alias Analysis
WPA	Whole Program Analysis

Attachments

`Bloomap.zip`
Implementation of Bloomaps in C++. Also available online on github:
`http://github.com/Krakonos/Bloomap`

`gcc-ipa-kpta.diff`
Patch to GCC implementing IPA KPTA. Also available online on github:
`http://github.com/Krakonos/kgcc`

`cgstat`
A tool for measuring resource usage using Linux Control Groups.

`thesis.pdf`
Digital version of this thesis.