Compiler optimizations based on call-graph flattening

Master of Science in Telecommunication Engineering
Third School of Engineering: Information Technology
Politecnico di Torino

supervisor
prof. Silvano Rivoira

candidate
Carlo Alberto Ferraris

July 6th, 2011

To my parents, kins, friends
and all that is good under the sun.

Contents

Nomenclature

Σ	CGF dispatch function
BB	Basic Block
CFG	Control-Flow Graph
CGF	Call Graph Flattening
CPS	Continuation-Passing Style
CSE	Common Sequence Elimination
EH	Exception Handling
IPO	Inter-Procedural Optimization
IR	Intermediate Representation
JIT	Just-In-Time [compilation]
LLVM	Low Level Virtual Machine
LTO	Link-Time Optimization
PGO	Profile-Guided Optimization
SEH	Structured Exception Handling
SSA	Static Single Assignment
TCO	Tail-Call Optimization
VM	Virtual Machine
ϕ	SSA phi node

List of Algorithms

List of Figures

List of Tables

1 Introduction

1.1 Increasing complexities

The current trend towards an ever-increasing integration and convergence of heterogeneous systems and platforms inevitably causes an exponential growth of the complexity of their underlying implementation logic.

At the same time, another strong trend is the ongoing research for solutions to improve the energetic efficiency of IT systems, be it because of dependency on battery power - an increasingly common scenario in mobile systems - or because of power density problems - something that must be taken into careful consideration in data centers.

These two contrasting trends raise stronger than ever the need for effective optimizations of compiled code: increasingly bigger quantities of code are to be executed as efficiently as possible.

In this thesis we analyze in depth these problems and then we proceed to propose and analyze optimizations based on call-graph flattening (CGF), a well-known code transformation technique.

The thesis is organized as follows: the remainder of this chapter gives a high-level introduction to the problems outlined above and to some general concepts that will be useful in later chapters. The second chapter presents in detail CGF and its effects, especially with respect to the issues highlighted in chapter 1. The third chapter presents the implementation effort undertaken to test the proposed transform and the fourth chapter analyzes the results of code optimized by CGF. Chapter 5, finally, concludes the thesis by presenting a summary of the work done and of possible future directions.

1.1.1 Abstractions

The growing complexity of the systems mentioned in the previous section requires abstractions to hide and keep under control the underlying logic.

While abstractions are good practice because they allow modularization, code reuse and verifiability they are also a source of overhead in many different terms: code size, memory usage and execution speed. A good ad-hoc implementation will generally be faster and use less resources than an implementation built on abstractions, but the latter will generally be faster for the programmer to write, since hopefully the abstractions will take care of some of the details.

To reap the best of both worlds we would like to be able to create abstractions general enough to be useful for a wide variety of uses but, at the same time, that can be highly specialized to resemble ad-hoc implementations.

It is important to understand that by abstractions we don't just mean the abstractions provided by a library or framework of some sort, but by all the universe of possibilities offered by interpreters, scripting languages, virtual machines and so on, as well as techniques such as design-by-contract or structured programming.

Scripting languages and virtual machines are indeed becoming more and more common because they allow easier and faster application development thanks to higher-level abstractions (dynamic typing, automating resource management, sandboxing semantics, etc.). All of them are powered by some form of interpreter that execute either source-, byte- or object-code. Since interpretation overhead compared to native code is quite high, just-in-time (JIT) compilation is increasingly used to reduce the difference in performance between interpreted and native code. Essentially, JIT compilation takes at runtime frequently-executed interpreted code and turns it into native machine code that can be executed without further interpretation overheads.

1.1.2 Cross-language barriers

The proliferation of different programming languages, each with their strength and peculiarities, poses a number of additional issues when we need to obtain interoperability between them (i.e. sharing data and performing cross-language calls).

This fact brings along a great number of untackled possibilities because of the difference in semantics between different languages. To minimize the number of possible issues, glue code between languages is generally very slow because it must account for all the differences in semantics between the glued languages.

The technique that is discussed in this thesis, call-graph flattening, could prove really powerful in this case because, provided that the two languages target the same IR, it would allow inferring static properties across language boundaries.

Examples of concrete situations where such capabilities would be really interesting include marshalling operations and calling convention adaptation.

Marshalling is the operation of converting and adapting data from a system, framework or programming language to another. This happens for example when sharing data across language or framework boundaries and may involve more than just converting data types: most often different frameworks use different resource management policies (e.g. manual or automatic garbage collection, call-based or stack-based, etc.). Being able, for example, to infer the life-span of an object that is shared between frameworks might avoid unnecessary load on the automatic garbage collector.

1.2 Resource scarcity

In recent years there has been a growing push towards a pervasive presence of processing and communication devices. This trend has been given the name of ubiquitous computing[1].

[1] many other names have been proposed but no clear winner has stood out yet: these include pervasive computing and physical computing

At their core, all models of ubiquitous computing share a vision of small, inexpensive, robust networked processing devices, distributed at all scales throughout everyday life and generally turned to distinctly common-place ends. For example, a domestic ubiquitous computing environment might interconnect lighting and environmental controls with personal biometric monitors woven into clothing so that illumination and heating conditions in a room might be modulated, continuously and imperceptibly. Another common scenario posits refrigerators aware of their suitably tagged contents, able to both plan a variety of menus from the food actually on hand, and warn users of stale or spoiled food.

Ubiquitous computing presents challenges across computer science: in systems design and engineering, in systems modeling, and in user interface design. Contemporary devices that lend some support to this latter idea include mobile phones, digital audio players, radio-frequency identification tags (RFID) and smart objects[1]. Emerging systems in this field include wireless sensor networks, wearable computers, digital systems empowering nanorobotics, etc.

Whereas some of the above systems might have considerable amount of processing power (e.g. mobile phones) others have very tight allowances for the amount of code and computations. Examples of such systems include RFIDs, wireless sensors, etc. At the same time, many functional requirements are placed on these very same devices, such as the ability to run standard network stacks (IP), to employ reusable software components and to provide the typical high-level abstractions available in modern operating systems. This trend is particularly well illustrated in some mobile operating systems for smartphones (e.g. webOS) where most of the user interaction is managed by software written in HTML+CSS+Javascript.

All of this is made even harder when considering the implicit distributed nature of some of such systems. This means accounting for potentially high failure rates (mainly due to communication errors but also to other factors as well) and for the huge amount of security implications of such systems (where the presence of malfunctioning or byzantine[2] nodes can potentially be way higher). At the same time some applications (e.g. wireless sensors) require that nodes are highly available and fail safe, either because they provide crucial data or because they are placed in places where servicing is impractical or downright impossible. This, in turn, requires thorough input validation, continuous self testing capabilities and telemetry support.

The main problem with such systems is obviously that they tend to be battery powered. As such, efficient code is a necessity because it allows doing the same computations faster or with lower energy consumption (and this might as well allow cost savings, because a less powerful processor or battery can be used instead).

Finally, a resource that it's not really bound to the platform itself but rather at its development is development time - i.e. the time spent developing the product, that must be as small as possible to decrease product time-to-market and reduce costs. To this end a commonly used solution is code reuse that, while very good at solving the problem, is also a source of additional runtime overhead, because code general enough to be reused

[2] a byzantine node is a node that willingly works to disrupt the functionality of other nodes in the network or system

will hardly be tuned for the application being developed.

1.3 Optimizations

To alleviate the problems outlined in the previous sections, compilers have gained automatic code optimizations to increase the performance of compiled code.

More precisely, compiler optimization is the process of tuning the output of a compiler to minimize or maximize some attributes of an executable computer program. The most common requirement is to minimize the time taken to execute a program[3] but others, such as minimizing the amount of memory occupied, the size of the compiled code or the energy consumed are possible as well. These last requirements are indeed quickly regaining importance because of the growth of mobile, heterogeneous and pervasive computing.

Compiler optimization is generally implemented using a sequence of *optimizing transformations*, algorithms which take a program and transform it to produce a semantically equivalent output program that uses less resources.

It has been shown that some code optimization problems are NP-complete, or even undecidable. Even in practical implementations, optimization is generally a very CPU- and memory-intensive process. Therefore, factors such as the programmer's willingness to wait for the compiler to complete its task and memory limitations place upper limits on the optimizations that a compiler implementor might provide. Because of all these factors, optimization rarely produces optimal output in any sense, and in fact an "optimization" may impede performance in some cases; rather, they are heuristic methods for improving resource usage in typical programs[2].

In the remainder of this section we briefly survey some key concepts in compiler optimizations that will be used in the following chapters.

1.3.1 Taxonomy of compiler optimizations

To a large extent, compiler optimization techniques have the following themes[2], which sometimes conflict:

Optimize the common case The common case may have unique properties that allow a fast path at the expense of a slow path. If the fast path is taken most often, the result is better overall performance.

Infer code and data properties Propagate knowledge about the code and the expected inputs to create code optimized for a restricted range of conditions.

[3] execution speed is not only attained by choosing the shortest instruction sequence but is also a function of memory access patterns, memory bandwidth, pipelining issues, locking and synchronization, etc.

Less and simpler code Remove unnecessary computations and intermediate values. Replace complex or expensive operations with simpler ones[4]. Less work for the CPU, cache, and memory usually results in faster execution. Alternatively, in embedded systems, less code brings lower power consumption and lower product cost.

Avoid redundancy Reuse results that are already computed and store them for use later, instead of recomputing them.

Avoid code branching Jumps and function calls interfere with the prefetching of instructions, thus slowing down code. Using inlining or loop unrolling can reduce branching, at the cost of increasing binary file size by the length of the repeated code. This tends to merge several basic blocks into one. Even when it makes performance worse, programmers may eliminate certain kinds of jumps in order to defend against side-channel attacks.

Exploit the memory hierarchy Accesses to memory are increasingly more expensive for each level of the memory hierarchy, so place the most commonly used items in registers first, then caches, then main memory, before going to disk. Code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference.

Parallelize Reorder operations to allow multiple computations to happen in parallel, either at the instruction, memory, or thread level.

1.3.1.1 Inter-procedural optimizations

Interprocedural optimization (IPO) is a set of optimizations particularly useful when compiling programs containing many frequently used functions of small or medium length. IPO differs from other compiler optimization because it analyzes the entire program; other optimizations look at only a single function, or even a single block of code.

IPO seeks to reduce or eliminate duplicate calculations across functions, inefficient use of memory, and to simplify iterative sequences such as loops. If there is a call to another routine that occurs within a loop, analysis may determine that it is best to inline the call[3]. Additionally, functions may be reordered for better memory layout and locality[4].

1.3.1.2 Vectorizing and parallelizing optimizations

Automatic vectorization and parallelization refers to converting sequential code into vectorized or multi-threaded code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine or on special-purpose processors (GPUs, ASICs). The goal of automatic parallelization is to relieve programmers from the tedious and error-prone manual parallelization process[5]. Though the quality of automatic

[4] For example, replacing division by a constant with multiplication by its reciprocal, or using loop induction variable analysis to replace index multiplication with addition.

parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation[6].

The programming control structures on which such transformations place the most focus are loops, because, in general, most of the execution time of a program takes place inside some form of loop. A parallelizing compiler tries to split up a loop so that its iterations can be executed on separate processors concurrently.

Recently, due to the trends towards massively parallel architectures (such as GPUs), attention has been also shifting towards more general control-flow constructs[7].

1.3.2 Static code analysis

Static code analysis[5] is a set of techniques to analyze the properties of a computer program without executing it.

Examples of such properties include determining the possible values of a certain variable at a certain point in the program or determining unreachable code (i.e. code that can never be executed)[8].

These analysis are important because they allow compilers to perform smarter optimizations. Consider for example the case where we are accessing the elements of an array: if static analysis can prove that the code will never try to access an item outside the array (e.g. because we are iterating over a known range inside the array) the compiler can remove the bounds checking for all those accesses.

Another important use of static analysis is checking the code for correctness and vulnerabilities: static analysis can detect dangerous operations such as reading from uninitialized memory, dereferencing *NULL* pointers, potential buffer overruns and, more generally, cases of undefined behaviors.

1.3.3 Intermediate representation

In many cases during compilation the source code is stored in an intermediate form that the compiler/interpreter can efficiently use respectively to produce object code or to run the program.

These intermediate representations have a number of interesting properties that facilitates the work of the compiler e.g. in SSA a "variable" is always assigned a value exactly once so that use-def chains are explicit.

1.3.3.1 Call graph

A call graph is a directed graph that represents calling relationships between subroutines in a computer program. Specifically, each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g. Thus, a cycle in the graph indicates recursive procedure calls.

[5] also known as static program analysis

Call graphs are a basic program analysis result that can be used for human understanding of programs, or as a basis for further analyses, such as an analysis that tracks the flow of values between procedures. One simple application of call graphs is finding procedures that are never called.

Call graphs can be dynamic or static. A dynamic call graph is a record of an execution of the program, e.g., as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is undecidable, so static call graph algorithms are generally over-approximations. That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

Call graphs can be defined to represent varying degrees of precision. A more precise call graph more precisely approximates the behavior of the real program, at the cost of taking longer to compute and more memory to store. The most precise call graph is fully context-sensitive, which means that for each procedure, the graph contains a separate node for each call stack that procedure can be activated with. A fully context-sensitive call graph can be computed dynamically easily, although it may take up a large amount of memory. Fully context-sensitive call graphs are usually not computed statically, because it would take too long for a large program. The least precise call graph is context-insensitive, which means that there is only one node for each procedure.

With languages that feature dynamic dispatch, such as Java and C++, computing a static call graph precisely requires alias analysis results. Conversely, computing precise aliasing requires a call graph. Many static analysis systems solve the apparent infinite regress by computing both simultaneously.

1.3.3.2 Control-flow graph (CFG)

The CFG is a representation, using graph notation, of all code paths that might be traversed through a program during its execution. It differs from the call graph because the CFG works at the basic block (BB) level, whereas the call graph works at the function level.

In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

The CFG is essential to many compiler optimizations and static analysis tools. For example, reachability can be computed based on it. If a block/subgraph is not connected from the subgraph containing the entry block, that block is unreachable during any execution, and so is unreachable code; it can be safely removed. If the exit block is unreachable from the entry block, it indicates an infinite loop. Again, dead code and some infinite loops are possible even if the programmer didn't explicitly code that way: optimizations like constant propagation and constant folding followed by jump threading

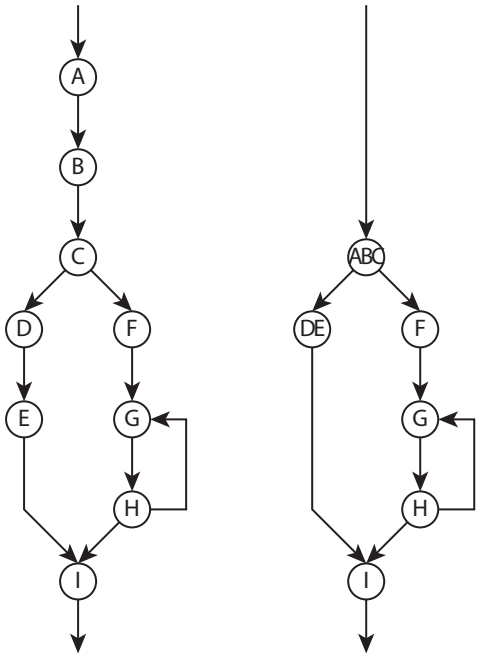

Figure 1.1: An example of CFG
original version (left), reduced version (right)

could collapse multiple basic blocks into one, causing edges to be removed from a CFG, etc., thus possibly disconnecting parts of the graph.

The CFG also plays an important role at link-time because the number of times an edge is crossed during program execution can be used to decide how to lay out the object code to minimize cache misses and cache pollution.

A number of terms are used to designate blocks or set of blocks having important properties w.r.t. a given block. The following list enumerates the most common ones:

Predecessor block Block through which the control flow can reach another block. In figure 1.1:

- A has no predecessors
- A and B are the predecessors of C
- A, B, C, D, E, F, G, H is the set of the predecessors of I
- A, B, C, F, G, H is the set of the predecessors of G[6]

[6] in particular, G is a predecessor of itself: this is an easy way to detect loops

- ...

Immediate predecessor block Block through which the control flow can reach directly another block. An immediate predecessor of block X has an outgoing edge towards X. In figure 1.1:

- A has no immediate predecessors
- B is the immediate predecessor of C
- E and H are the immediate predecessors of I
- H and F are the immediate predecessors of G
- ...

Successor block Block that the control flow can reach through the current block. In figure 1.1:

- B, C, D, E, F, G, H, I is the set of the successors of A
- D, E, F, G, H, I is the set of the successors of C
- I has no successors
- G, H, I is the set of the successors of G[7]
- ...

Immediate successor block Block through which the control flow can reach directly another block. An immediate successor of block X has an incoming edge coming from X. In figure 1.1:

- B is the immediate successor of A
- D and F are the immediate successors of C
- I has no immediate successors
- H is the immediate successor of G
- ...

Entry block Block through which all control flows enter the graph. The entry block has no predecessors. In figure 1.1 A is the entry block.

[7] similarly to the predecessor case, G is a successor of itself

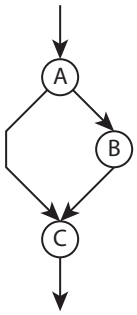

Figure 1.2: Example of a CFG with a critical edge

Exit block Block through which all control flows leave the graph. The exit block has no successors. In figure 1.1 I is the exit block.

Back edge An edge that points to a predecessor, i.e. an ancestor in a depth-first traversal of the graph. In figure 1.1 the edge going from H to G is a back edge.

Critical edge An edge which is neither the only edge leaving its source block, nor the only edge entering its destination block. These edges must be split (a new block must be created in the middle of the edge) in order to insert computations on the edge without affecting any other edges. In figure 1.2 the edge from A to C is critical edge.

Abnormal edge An edge whose destination is unknown. Exception handling constructs can produce them. These edges tend to inhibit optimization.

Impossible edge An impossible edge[8] is an edge which has been added to the graph solely to preserve the property that the exit block postdominates all blocks. It cannot ever be traversed.

Dominator A block dominates another block if every path from the entry that reaches the latter has to pass through the former. The entry block dominates all blocks. In figure 1.1:

- the set of dominators of block G is A, B, C, F
- the set of dominators of block I is A, B, C
- C dominates blocks D, E, F, G, H, I

[8] also known as fake edge

- F dominates blocks G, H
- ...

Postdominator A block postdominates another block if every path from the latter to the exit block has to pass through the former. The exit block postdominates all blocks. In figure 1.1:

- the set of postdominators of block B is C, I
- the set of postdominators of block F is G, H, I
- G postdominates block F
- I postdominates blocks A, B, C
- ...

Immediate (post)dominator A block immediately dominates another block if the former dominates the latter, and there is no intervening block P such that the former dominates P and P dominates the latter. In other words, the former is the last dominator on all paths from entry to the latter. Each block has a unique immediate dominator. In figure 1.1:

- C immediately dominates blocks D, F, I
- G immediately dominates block H
- ...

The immediate postdominator is analogous to the immediate dominator but for the postdominator case.

(Post)dominator tree An ancillary data structure depicting the dominator relationships. There is an arc from a block to another block if the former is an immediate dominator of the latter. This graph is a tree, since each block has a unique immediate dominator. This tree is rooted at the entry block. Can be calculated efficiently using Lengauer-Tarjan's algorithm.

Similarly, the postdominator tree is analogous to dominator tree but for the postdominator case. This latter tree is rooted at the exit block.

Loop header Sometimes called the entry point of the loop, a dominator that is the target of a loop-forming back edge. Dominates all blocks in the loop body. In figure 1.1 block G is a loop header.

Loop pre-header Suppose block M is a dominator with several incoming edges, some of them being back edges (so M is a loop header). It is advantageous to several optimization passes[9] to create a new block P, initially containing only a jump to M (so that P is the immediate dominator of M) and moving all incoming edges from outside the loop so that they point to P. In figure 1.1 block F can be considered a loop pre-header.

1.3.3.3 Static single assignment (SSA)

SSA is a property of an intermediate representation (IR), which says that each variable is assigned exactly once. Existing variables in the original IR are split into versions, new variables typically indicated by the original name with a subscript in textbooks, so that every definition gets its own version. In SSA form, use-def chains are explicit and each contains a single element.

SSA is formally equivalent to a well-behaved subset of CPS (excluding non-local control flow, which does not occur when CPS is used as intermediate representation), so optimizations and transformations formulated in terms of one immediately apply to the other.

The primary usefulness of SSA comes from how it simultaneously simplifies and improves the results of a variety of compiler optimizations, by simplifying the properties of variables. For example, consider this piece of code:

```
1 a = 3
2 a = 0
3 b = a
```

A person can see immediately that the first assignment is not necessary, and that the value of a being used on line 3 comes from the second assignment of a on line 2. A program would have to perform reaching definition analysis to determine this. But if the program is in SSA form, both of these are immediate:

```
1 a_1 = 3 // a_1 is never used
2 a_2 = 0
3 b_1 = a_2
```

Phi nodes Since, by definition, variables in SSA form have exactly one constant definition, a problem arises when the value of a variable depends on branches in the code. Let's consider the following example:

```
1 int a = b > 0 ? b : 0;
```

We can see that the value of a comes from one of two different definitions depending on the value of b, thereby hindering the explicitness of the use-def chain.

To solve this problem, a new kind of node is introduced in the SSA representation. These special nodes are called ϕ nodes[10] and they are used in BBs having multiple

[9] chiefly, invariant code motion

[10] also called phi nodes or phoney nodes

predecessors where the value of a variable depends on which predecessor the control is coming from. More formally:

$$\phi(V_1, F_1, V_2, F_2, ..., V_n, F_n) = \begin{cases} V_1 & \textit{if control comes from } F_1 \\ V_2 & \textit{if control comes from } F_2 \\ ... & ... \\ V_n & \textit{if control comes from } F_n \end{cases}$$

It should be noted that if the value is the same on all incoming edges to a BB, the ϕ node is not needed: this is the case when the value has been set in one of the dominators of the BB. For example, in figure 2.1, if value V_B has been defined in BB B, we don't need the corresponding ϕ nodes in G and I because they are both dominated by$B$.

Conventionally ϕ nodes are placed at the beginning of the BB, before any other instruction.

Live variables A variable is live at a certain point p of the program if it has been defined before p (i.e. a value has been written to it) and there are reads (uses) of the variable in or after p. Intuitively a live variable can be thought as a variable whose current value will still be needed in the future.

If a variable is not live it is considered dead. A dead variable is a variable whose value is not needed anymore. As such any resource (register, stack, heap) used to store the variable can be reclaimed and used for something else.

2 Call graph flattening

Moving from the analysis done in chapter 1 we think it should be interesting trying to apply a well-known CFG transformation technique - call graph flattening (CGF) - for optimizations purposes.

This transformation can be thought as a global inlining pass (i.e. all function calls are inlined) followed by a global macro compression pass (i.e. all duplicated code - especially duplicated inline function bodies - is merged and joined by jumps). Simplifying we could say that all *call* and *return* instructions are transformed into *jump* instructions, while statically keeping track of the variables that are live at each point in the program.

Formally, this means creating a global function (called Σ-function or dispatch function) that contains the bodies of all the functions in the module[1] being transformed and replacing all calls between them with jumps.

A graphical representation of this transformation is given in figures 2.1 and 2.2: we start from two functions X and Y, the former containing a call to the latter. Let's assume, for the sake of the discussion, that:

- basic block D only contains a conditional branch that we know will always lead to E if coming from B
- that Y is called most of the time from X and only seldomly from other functions (not shown)

After CGF (figure 2.1, middle) we obtain a single function (Σ) containing the bodies if both X and Y. The call to Y in B has been replaced by a jump and B itself has been split in two halves at the callsite. After optimization (figure2.1, bottom), given the first assumption above, the edge (B, D) has been transformed into (B, E). After block reordering and code emission (figure 2.2), taking into consideration the second assumption, we can see that part of function Y has been inlined in X even if it is still being used by the rest of function Y.

Intuitively, the aim of this is to remove the distinctions between intra-procedural and inter-procedural optimizations by merging the bodies of all functions in the module. This, in turn, should allow further optimizations to take place.

[1] or program, in which case the Σ-function coincides with main() and a number of interesting properties apply which will be discussed in section 2.4

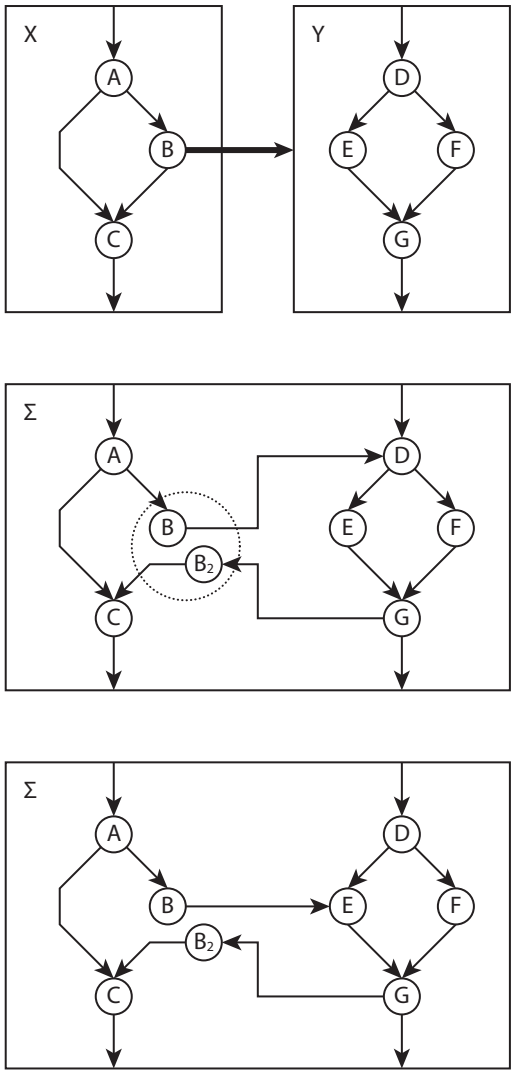

Figure 2.1: Example of CGF transformation
Original CFG with two functions X and Y (top), CGF-transformed version, the dotted circle contains the result of splitting basic block B at the callsite (middle), possible optimization result if D were to contain only a conditional branch that we know will always jump to E if coming from B, note that the edge from D to E has now become a critical edge (bottom).

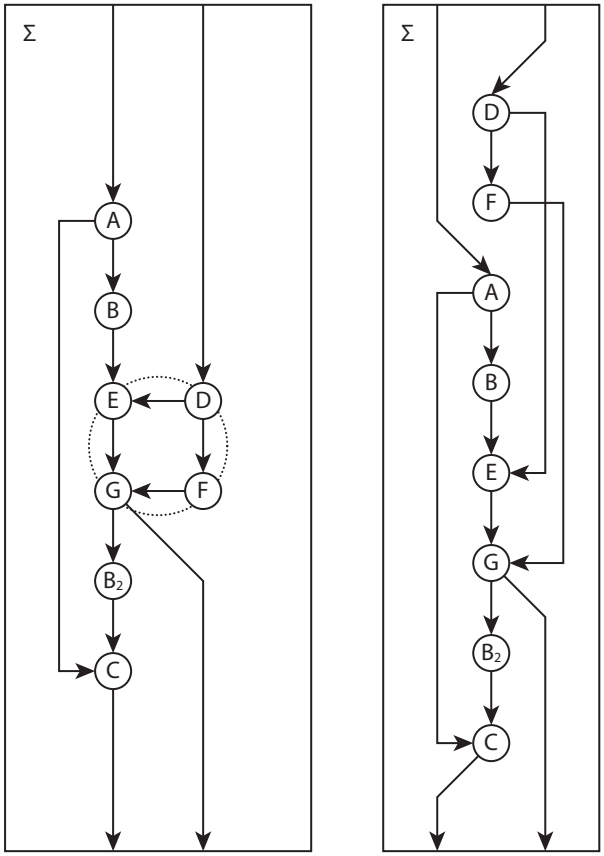

Figure 2.2: Example of CGF-tranformed function layout
The same function of figure 2.1 after block reordering (left, the dotted circle highlights function Y) and after code emission (right). E and F are placed inline between B and B_2 because we assumed that the vast majority of calls of to function Y comes from function X. In this way blocks A, B, E, G, B_2 and C are contiguous with a single conditional branch in G. We have therefore obtained something close to the result of inlining Y in X but without any code duplication.

As a slightly more concrete example consider the following snippet of code:

```
1  int foo(int i) {
2    return bar(i * i);
3  }
4
5  int bar(int i) {
6    return i - 1;
7  }
8
9  int baz(int i) {
10   return bar(1 << i);
11 }
```

Running CGF would then yield something similar[2] to:

```
1  dispatch(func, param1) {
2    dispatch_start:
3      // jump to the entry block of the inlined functions
4      // in this case, func can be either dispatch_foo or
5      // dispatch_bar
6      goto func;
7
8    dispatch_foo:
9      // after_foo holds the address of the basic block to
10     // jump when returning from foo
11     after_foo = dispatch_end;
12     // the parameter param1 is put in variable i of function
13     // foo, renamed foo_i to avoid naming collisions
14     foo_i = param1;
15     // now jump to the body of function foo
16     goto foo;
17
18   foo:
19     foo_1 = foo_i * foo_i;
20     // after_bar contains the address of the basic block to
21     // jump to when returning from bar. foo_bb1 is a newly
22     // created basic block obtained by splitting the body of
23     // function foo immediately after the call to bar
24     after_bar = foo_bb1;
25     // the parameter i of function bar
```

[2] in this and many other examples we will assume that the dispatch function receives as first parameter the address of one of the labels contained inside it: this is not allowed by the C programming language but can be done easily when the trasformation operates directly on the compiler IR (for more details see section 3.3)

```
26      bar_i = foo_1;
27      // jump right to the body of function bar
28      goto bar;
29
30    foo_bb1: // basic block to return to after
31      retval = retval_bar;
32      goto after_foo;
33
34    dispatch_bar:
35      after_bar = dispatch_end;
36      bar_i = param1;
37      goto bar;
38
39    bar:
40      retval_bar = bar_i - 1;
41      retval = retval_bar;
42      goto after_bar;
43
44    dispatch_baz:
45      after_baz = dispatch_end;
46      foo_i = param1;
47      goto baz;
48
49    baz:
50      baz_1 = 1 << baz_i;
51      after_bar = baz_bb1;
52      bar_i = baz_1;
53      goto bar;
54
55    baz_bb1:
56      retval = retval_bar;
57      goto after_baz;
58
59    dispatch_end:
60      return retval;
61  }
```

if we reorder the order of the basic blocks, we get:

```
1   dispatch(func, param1) {
2     dispatch_start:
3       goto func;
4
5     dispatch_bar:
```

```
6     after_bar = dispatch_end;
7     bar_i = param1;
8     goto bar;
9
10  dispatch_foo:
11    after_foo = dispatch_end;
12    foo_i = param1;
13    goto foo;
14
15  foo:
16    foo_1 = foo_i * foo_i;
17    after_bar = foo_bb1;
18    bar_i = foo_1;
19    goto bar;
20
21  dispatch_baz:
22    after_baz = dispatch_end;
23    baz_i = param1;
24    goto baz;
25
26  baz:
27    baz_1 = 1 << baz_i;
28    after_bar = baz_bb1;
29    bar_i = baz_1;
30    goto bar;
31
32  bar:
33    retval_bar = bar_i - 1;
34    retval = retval_bar;
35    goto after_bar;
36
37  foo_bb1:
38    retval = retval_bar;
39    goto after_foo;
40
41  baz_bb1:
42    retval = retval_bar;
43    goto after_baz;
44
45  dispatch_end:
46    return retval;
47 }
```

and this can be simplified by standard optimizations to:

```
1  dispatch(func, param1) {
2    jump func;
3
4    dispatch_baz:
5      param1 = 1 << param1;
6      goto dispatch_bar;
7
8    dispatch_foo:
9      param1 = param1 * param1;
10
11   dispatch_bar:
12     retval = param1 - 1;
13
14   return retval;
15 }
```

Whenever a function is used, a certain amount of extra time is required for the CPU to carry out the call: it must save the registers used by itself, store the function arguments in the appropriate registers and/or stack locations, jump to the start of the function (bringing the appropriate virtual memory pages into physical memory or the CPU cache if necessary), begin executing the code. When the callee terminates, the inverse process has to be done (restoring the registers, reading the return value, jumping to the instruction following the call). This additional work is referred to as function-call overhead[9].

Function-call overhead can be partially or completely avoided by inlining the callee in the caller, i.e. replacing the call instruction with the body of the callee. Beside avoiding the function-calling overhead this also allows further optimizations to take place because the compiler is now free to specialize the inlined callee.

Unfortunately, inlining has a few serious drawbacks as well. Chiefly, inlining duplicates[3] the code of the callee, leading to code size (and memory usage) increase. This, in turn, might slow down execution because the code is less likely to fit in the CPU cache, leading to cache misses that - especially on modern processors - are much more costly performance-wise than function-call overhead[10]. For this reason, inlining can only be used sparingly[9, 11, 12].

CGF on the other hand allows to avoid these drawbacks: in the example above we can see that CGF has effectively yielded a function that contains an inlined version of all original functions without any code duplication and with minimal overhead (one indirect branch and one unconditional branch), at the same time avoiding most of the machinery associated with function calls, i.e. explicit stack frame management and function prologue/epilogue insertion. What's important is that all three functions can

[3] there is one important case when this does not hold and is, in fact, the only case when inlining has no drawbacks at all: when a function is called from a single callsite in the whole program the original function can be removed altogether and its body inlined in the caller.

still be executed separately. It is indeed enough to create a forwarding wrapper[4] for each function to obtain backward binary compatibility with external callers:

```
1  int foo(int i) {
2    return dispatch(dispatch_foo, i);
3  }
4
5  int bar(int i) {
6    return dispatch(dispatch_bar, i);
7  }
8
9  int baz(int i) {
10   return dispatch(dispatch_baz, i);
11 }
```

This example is obviously limited because it places a number of undesirable constraints on the functions being transformed (namely, same signature) and also doesn't address a number of possibilities that might happen in real world code (such as direct or indirect recursion, indirect function calls, etc.). We'll show in chapter 2.4 how to relax these constraints.

2.1 Previous uses

CGF is a well known technique first mentioned in 1977[15].

CGF has been used in the past mostly as a technique to obfuscate compiled code rather than to optimize it[16, 17, 18, 19]. Its effect is indeed that of completely destroying the native function-level structure of the program by merging the whole program in a single global function. It must be noted, though, that previous attempts were mostly implemented as source-to-source transformations (mostly C-to-C) deliberately targeted at obfuscating the program structure to prevent reverse-engineering.

Beside the fact that the objective was not performance but obfuscation, the optimizations efforts of the compiler were furthermore hindered by the spurious semantics of the C language. By implementing the transformation as a IR-to-IR transformation, it is possible to have greater control on the output and to avoid introducing unwanted obstacles for further optimizations.

More recently, CGF has also been proposed as a memory-optimization transformation targeted especially at embedded applications in [20]. Targeting a 8-bit micro-controller the authors were able to reduce the average stack usage by 20% and reducing CPU usage by 3% with a C-to-C transformation and some custom optimizations. At the same time, code size increased by 13%.

Due to its nature, such a transformation is obviously naturally suited to be run at link-time rather than at compile time: by including in the transformation all the modules

[4] also known as a trampoline[13] or shim[14]

of the program as well as the statically-linked libraries the most extensive optimizations can be achieved. Ideally, this transformation could be even run at runtime by the dynamic linker; this would allow very interesting possibilities for language-based systems (see section 2.3.3.6).

2.2 Related concepts

A number of other techniques are somewhat similar to CGF. This may be the case because either they are a more general/particular case of CGF or they might provide similar results in radically different ways.

In this section we highlight some of them and their relation with CGF.

2.2.1 Continuation-passing style (CPS)

In functional programming, continuation-passing style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation. The term was used for the first time in 1975 inside the first specification of the Scheme programming language[21].

Instead of returning values as in the more familiar direct style, a function written in continuation-passing style takes an explicit continuation argument, i.e. a function which is meant to receive the result of the computation performed within the original function. Similarly, when a subroutine is invoked within a CPS function, the calling function is required to supply a procedure to be invoked with the subroutine's return value. Expressing code in this form makes a number of things explicit which are implicit in direct style. These include: procedure returns, which become apparent as calls to a continuation; intermediate values, which are all given names; order of argument evaluation, which is made explicit; and tail calls, which simply is calling a procedure with the same continuation that was passed to the caller, unmodified.

Despite their formal difference, SSA and CPS forms are equivalent[22]. Programs can be automatically transformed[5] from direct style to CPS. Functional and logic compilers often use CPS as an intermediate representation where a compiler for an imperative or procedural programming language would use SSA[23].

CGF is conceptually similar to CPS: instead of creating an explicit continuation composed of the function to be called at the end of the callee and the variables to be passed to it, we forward to the callee the address of the basic block to jump to when returning and all the live values that will be needed. In particular, for non recursive code, CGF and CPS are equivalent[24].

2.2.2 Tail calls

A tail call is a subroutine call that happens inside another procedure and that produces a return value, which is then immediately returned by the calling procedure. The call site is

[5] technically there are constructions in CPS that cannot be translated to SSA, but they do not occur in practice

then said to be in tail position, i.e. at the end of the calling procedure[6]. If a subroutine performs a tail call to itself, it is called tail-recursive. This is a special case of recursion.

Tail calls are significant because they can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is not needed any more, and it can be replaced by the frame of the tail call, modified as appropriate. The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called tail call elimination, or tail call optimization[15].

Traditionally, tail call elimination is optional. However, in functional programming languages, tail call elimination is often guaranteed by the language standard, and this guarantee allows using recursion, in particular tail recursion, in place of loops[25]. In particular, it is worth noting that all function calls in CPS are tail calls.

2.2.3 setjmp/longjmp

setjmp and *longjmp* are complementary functions defined in the C standard library (*setjmp.h*) to provide "non-local jumps", i.e. control flow that deviates from the usual subroutine call and return sequence.

A typical use of *setjmp*/*longjmp* is implementation of an exception mechanism that utilizes the ability of *longjmp* to reestablish the program/thread state, even across multiple levels of function calls. A less common use of *setjmp* is to create syntax similar to coroutines[26, 27].

Consider the following example where a simple try-catch construct is shown:

```
1 try {
2   // do something
3 } catch (exception) {
4   // handle exception
5 }
```

with *setjmp*/*longjmp* it is possible to achieve something similar:

```
1 int exception;
2 jmp_buf e;
3 if (!(exception = setjmp(e))) {
4   // do something
5 } else {
6   // handle exception (thrown using longjmp(e))
7 }
```

CGF is conceptually similar to the idea of non-local jumps but is rather different in its implementation: *setjmp* saves the execution state of the thread (mainly the stack pointer and program counter) at the moment of invocation in the *jmp_buf* and *longjmp* restores it by overwriting the stack pointer and program counter with the ones that were saved

[6] more properly, a call is a tail call if the set of live variables across the callsite is empty, something that is often true if the call is the last instruction before the return

by *setjmp* (i.e. without performing any proper stack unwinding - any stack unwinding should be performed manually after the *setjmp* has returned). CGF on the other hand operates by statically keeping track at compile time of all the values in the IR, delegating register allocation and stack management to the compiler backend.

Depending on the desired semantics, the set of values available to the exception handler (catch site) could either be the one available at the throwing site[7] or the one available at the call site inside the try block.

It's important to note that many other constructs such as continuations, generators, coroutines, cooperative multi-threading and the already mentioned exception handling are easily implemented using *setjmp*/*longjmp*. The reason why this is important is because we will see that CGF can be used to provide the same non-local jump semantics with lower overhead.

2.2.4 Code threading

The term threaded code refers to a compiler implementation technique where the generated code has a form that essentially consists entirely of calls to subroutines. The code may be processed by an interpreter, or may simply be a sequence of machine code call instructions.

One of the main advantages of threaded code is that it is very compact, compared to code generated by alternative code generation techniques. This advantage usually comes at the expense of slightly slower execution speed (usually only one machine instruction). However, sometimes there is a synergistic effect because more compact code better exploits instruction cache locality. A program small enough to fit fully in the processor cache may run faster than a larger program that suffers many cache misses[28].

2.2.5 Spaghetti code

Spaghetti code is a pejorative term for source code that has a complex and tangled control structure, either at the logic or syntax level, especially when using unstructured branching constructs (i.e. jumps). The name comes from the fact that program flow tends to look like a bowl of spaghetti, i.e. twisted and tangled.

This concept is related to CGF mostly because CGF, by merging all the functions together, potentially allows the compiler to mix together code pertaining to different functions by means of unstructured control flow (i.e. jumps instead of function calls). Obviously, in the case of CGF, this effect and its consequences are deliberate.

2.3 Applicability

In this section we present a number of concrete application examples of CGF.

[7] this would also allow resumable exceptions, i.e. the ability to resume execution at the instruction following the one that generated the exception, something that is not easily doable with *setjmp*/*longjmp*

2.3.1 Stack usage reduction

Normally, all live variables in registers are saved (pushed) on the stack before[8] calling the function and restored (popped) after the function has returned.

In CGF-transformed code, instead, the stack is used only when register spilling is inevitable or rematerialization is not profitable.

Also, since all transformed functions use the same stack frame, copies of the same values are never pushed onto the stack. For example, consider the following example:

```
1 int F(int a, int b) {
2   int c = a + b;
3   return G(a, c) + c;
4 }
5
6 int G(int a, int c) {
7   // do something
8 }
```

If we were to compile normally the above functions and take a snapshot of the stack during the execution of G, we would find something like[9] the state shown in figure 2.3. We can see that formal parameters a and c are duplicated because G has no way of knowing that the same values can be found in the stack frame of F. CGF on the other hand makes the compiler keep track of which values are available at all points in the program, and therefore avoids pushing duplicated values on the stack.

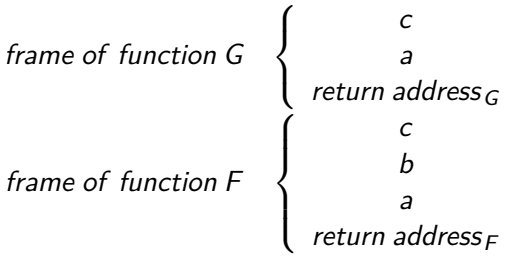

Figure 2.3: Contents of the stack at the breakpoint

While inside G the stack pointer points to $return\_address_G$. Upon returning from G, the return value is placed at the location of the stack pointer, the stack pointer is decremented to point to $return\_address_F$ and execution resumes from $return\_address_G$.

Obviously, this is a deliberately reductive example: once the code has been transformed by CGF there are many more possible optimizations.

[8] depending on the calling convention in use, the responsability of spilling the registers on the stack might be either of the caller's or the callee's (in most calling conventions is the caller's)

[9] the real layout of the stack is obviously dependent on the CPU architecture, ABI and the calling convention in use

2.3.2 Inlining replacement

Inlining is an optimization that replaces a function call site with the body of the callee. This optimization may improve time and space usage at runtime, at the possible cost of increasing the final size of the program (i.e. the binary file size)[3, 11, 12].

Ordinarily, when a function is invoked, control is transferred to its definition by a branch or call instruction. With inlining, control drops through directly to the code for the function, without a branch or call instruction. We have already seen in section 2 why inlining is useful. The primary cost of inlining is that it generally tends to increase code size. This may also decrease performance in some cases - for instance, multiple copies of a function may increase code size enough that the code no longer fits in the cache, resulting in more cache misses.

CGF, since it basically is a global inlining transformation, allows the compiler to reap the same benefits of inlining without the associated code duplication overhead.

2.3.3 Caller-contextual specialization

The great potential of CGF comes mainly from its caller-awareness. Normally functions are to be thought as independent black boxes, unaware of the dynamic call-graph that called them. In CGF, on the other hand, the control flow is statically known and as such many more contextual optimizations are possible.

2.3.3.1 Constant propagation

Since the whole CFG is statically known at compile time, we can use this knowledge to do caller-contextual constant propagation.

This in turn should allow fine-grained partial specialization. Consider the following example:

```
1  float A(float a) {
2    if (a<0)
3      return false;
4    a = round(a);
5    return log(a);
6  }
7
8  float B(float a) {
9    return B(a*a);
10 }
11
12 float C(unsigned short a) {
13   return B(a)
14 }
```

when A is called directly, the variable a can contain any value. But when A is called from B, we know that a can not be negative, so the check is redundant. Moreover when B is called from C we already know that a will be a (small) integer, so the call to *round* in A is useless.

Normally, short of inlining everything or creating specialized versions of A[10], we wouldn't be able to take advantage of this knowledge. With CGF, on the other hand, we get:

```
1  dispatch(func, param1) {
2    dispatch_start:
3      goto func;
4
5    dispatch_A:
6      a_A = param1;
7      after_A = dispatch_end;
8    A:
9      goto a < 0 ? A_1 : A_2;
10
11   A_1:
12     retval_A = false;
13     goto after_A;
14
15   A_2:
16     a = round(a);
17   A_3:
18     retval_A = log(a);
19     goto after_A;
20
21   dispatch_B:
22     B_a = param1;
23     after_B = dispatch_end;
24   B:
25     A_a = B_a * B_a;
26     after_A = B_1;
27     goto A;
28
29   B_1:
30     retval_B = retval_A;
31     goto after_B;
32
33   dispatch_C:
34     C_a = param1;
35     after_C = dispatch_end;
```

[10] the drawback being that both alternatives duplicate code

```
36    C:
37      B_a = C_a;
38      after_B = C_1;
39      goto B;
40
41    C_1:
42      retval_C = retval_B;
43      goto after_C;
44
45    dispatch_end:
46      return retval;
```

after subsequent optimizations, we should get:

```
1  dispatch(func, param) {
2    goto func;
3
4    dispatch_A:
5      goto a < 0 ? A_1 : A_2;
6
7    A_1:
8      return false;
9
10   dispatch_B:
11     a = a * a;
12   A_2:
13     a = round(a);
14   dispatch_C:
15     return log(a);
16 }
```

In the previous example, we showed that we can specialize the body of a function based on what precedes the function call. What's interesting is that this is also possible with what follows the callsite: a typical case when this might happen is if we don't use the return value of the callee. Consider for example the following function that replaces the content of a memory location with a new value and returns the old one.

```
1  int replace(int* ptr, int value) {
2    int tmp = *ptr;
3    *ptr = value;
4    return tmp;
5  }
```

In this example, loading the value from the memory location pointed to by *ptr* is only useful if the returned value is actually used. In case this isn't true, we would like to avoid

the load/store and replace it with a simple store. Let's assume we also have the following two functions:

```
1 int A(int* ptr, int value) {
2   return replace(ptr, value);
3 }
4
5 int B(int* ptr, int value) {
6   replace(ptr, value);
7   return value;
8 }
```

Transforming this example and optimizing it for the case where B is called more often than A would yield:

```
1  int dispatch(func, param1, param2) {
2    dispatch_start:
3      goto func;
4
5    dispatch_A:
6      retval = *param1;
7      goto replace;
8
9    dispatch_B:
10     retval = param2;
11   replace:
12     *param1 = param2;
13   dispatch_end:
14     return retval;
15 }
```

2.3.3.2 Stack unwinding

Stack unwinding is the act of returning from the called function, popping the top frame off of the stack, perhaps leaving a return value. With traditional functions, the frame must always be popped when returning from the callee[29].

There are many common idioms in use that could benefit by the kind of inter-procedural optimizations allowed by CGF.

As an example consider the following:

```
1 // ...
2 type *ptr = malloc(size_of_allocation);
3 if (ptr == NULL)
4   return NOT_ENOUGH_MEMORY;
5 // ...
```

While this is fairly straightforward, it's also quite wasteful because probably the implementation of *malloc* already performs a similar check in order to return *NULL* if the allocation fails. Indeed if we look at the code for FreeBSD's libc malloc(3)[30], we find the following[11]:

```
1 void *malloc(size_t size) {
2   void *ret;
3   // [various sanity checks]
4   ret = imalloc(size); // call to the internal allocator
5   if (ret == NULL)
6     errno = ENOMEM;
7   return ret;
8 }
```

As it can be seen, the check in our code is duplicated. Short of inlining the whole *malloc*, though, no optimization would ever be able to get rid of the duplicated check.

CGF on the other hand could easily allow the jump threading optimization to remove this redundancy ultimately yielding code that, right from the body of *malloc*, sets *errno* to *ENOMEM* and returns *NOT_ENOUGH_MEMORY* to the caller of our function above[12]. This can be thought as functionally equivalent to a *longjmp* without the associated overhead, because the compiler statically keeps track of which parts of the stack will be still needed after the jump and which, on the other hand, can be reused.

Even if the example provided might look like an unimportant corner case, consider that this could be applied to any function containing branches that are dependent on the result of another function.

Another important context where the term stack unwinding is used is during EH. This is discussed in section 2.3.6.3.

2.3.3.3 Virtual functions

Virtual function inlining has already been studied in the past[31]. This is normally accomplished by means of specializing a callsite to a virtual function like the following:

```
1 class BaseType { virtual void someMethod(); }
2 class DerivedType1 : public BaseType { void someMethod(); }
3 class DerivedType2 : public BaseType { void someMethod(); }
4
5 BaseType *b;
6 b->someMethod();
```

using either static or profiled information we can replace line 6 and create one or more fast paths that don't need dynamic dispatch:

[11] for clarity's sake the snippet shown here is a reduced version of the real implementation

[12] actually the imalloc implementation calls functions that do similar checks, so in principle all these level of indirection could be avoided

```
1 switch (typeof(b)) {
2   case DerivedType1: ((DerivedType1*)b)->someMethod(); break;
3   case DerivedType2: ((DerivedType2*)b)->someMethod(); break;
4   case BaseType:
5   default:           b->someMethod(); break; // virtual
6 }
```

This kind of transforms can greatly increase the performance of code using virtual functions[32]. Allowing virtual functions to be statically inlined by CGF could potentially allow even greater improvements, especially if coupled with macro compression (see section 2.3.5).

2.3.3.4 Contracts

When designing complex systems it is considered good practice to break down the system in well-scoped quanta that can be independently designed, verified and replaced (if necessary). This is to ensure that the design is not overwhelmed by complexity and that the implementation of each quanta can be carried on independently of each other (i.e. in parallel).

In software systems this is true as well: to make this happen each of these quanta (generally called components) have to be designed with well-defined semantics and behaviors. This generally means defining exactly what inputs and conditions each component should be able to handle, which output it should produce and how it should behave in case of errors.

Beside the actual implementation it is therefore sensible to explicitly code these semantics and behaviors, so that the compiler can enforce them both statically at compile-time and dynamically at run-time. This programming technique goes under the name of design-by-contract and relies heavily on the compiler to optimize away these checks.

Traditionally such kinds of checks were implemented via assertions, a C macro that was used to signal a condition that the programmer expected to be true at a certain point in a program. For example, the following would mean that two variables, A and B, are expected to hold the same value at the point of the program where the assert is placed:

```
1 assert(A == B);
```

The problem with this approach is that by default assertions are mainly intended as a debugging aid because they offer no way to recover from a failed assertion[13] and because they are ignored when building in non-debug configurations.

Other programming languages instead offer extensive support for contracts, by allowing to specify all kinds of conditions that a certain function should satisfy. Among these conditions, there are for example preconditions (i.e. conditions that must be true when

[13] if an assertion fails during debug the process is immediately terminated

the current function is called), postconditions (conditions that must be true when the function returns), invariants (conditions that are always true) and traditional assertions.

A typical example where contracts are useful is inside functions that operate on arrays. If the contract specifies that it is not allowed to access out of boundary elements, the caller is forbidden to ask for out-of-bound indexes[14] and the callee can completely avoid expensive bounds checking.

This allows great flexibility and power, because if these conditions can be statically proven to hold there is no need for runtime checks and therefore they can be omitted. If they can be statically proven to not hold, compilation is aborted and the problem reported immediately. If they can't be proven statically, then runtime checks are inserted in the code (generally coupled with SEH mechanisms).

This latter case is the most interesting for CGF, because as we will see in section 2.3.4 CGF allows, besides other optimizations, to pack together all scattered runtime checks and create fail-fast paths.

2.3.3.5 Managed code, interpreters and virtual machines

Interpreters and virtual machines (VMs) in their simplest form are generally implemented as some form of loop containing a switch:

```
1  interpreter_setup();
2  while (opcode = get_next_instruction())
3    interpreter_step(opcode);
4  interpreter_shutdown();
5
6  function interpreter_step(opcode) {
7    switch (opcode) {
8      case opcode_instruction_A:
9        execute_instruction_A();
10       break;
11     case opcode_instruction_B:
12       execute_instruction_B();
13       break;
14     // ...
15     default:
16       abort("illegal opcode!");
17   }
18 }
```

The interpreter loop simply translates the opcodes in memory to calls to the functions that implement each opcode. Generally, each one of these functions will have to perform a number of checks on the types and values of the parameters, on the state of the interpreter and also on the state of the system. These checks represent a sizable contribution to the

[14] attempts to do so, even non-deterministically, should generate a compile-time error

runtime overhead of interpreters: moreover, many of them could be removed if we could perform some kind of analysis on the code. For example, if we are doing some arithmetic operation on the result of another arithmetic operation, there would be no need to check the type of the value because we know that the result of an arithmetic operation can only be a number. These aspects will be analyzed in more detail in section 2.3.3.5.

For the sake of the following discussions, it is important to note that if we compile (with optimizations) the interpreter along the unrolled opcode stream we theoretically get a compiled version of the interpreted program:

```
1 interpreter_setup();
2 interpreter_step(opcode_1);
3 interpreter_step(opcode_2);
4 interpreter_step(opcode_3);
5 // ...
6 interpreter_shutdown();
```

This can be used, with the obvious differences, both to produce a statically compiled version ahead-of-time or a dynamically compiled version just-in-time.

2.3.3.6 Language-based systems

A language-based system is a type of operating system that uses language features to provide security, instead of or in addition to hardware mechanisms. In such systems, code referred to as the trusted base is responsible for approving programs for execution, assuring they cannot perform operations detrimental to the system's stability without first being detected and dealt with[33, 34]. A very common strategy of guaranteeing such operations are impossible is to base the system around a code whose well-defined semantics precludes dangerous constructs or unverifiable behaviors (e.g. Java bytecode).

Since language-based systems can assure ahead of time that programs can not do things that can damage the system (such as corrupting memory by dereferencing dangling pointers), it is possible for them to avoid expensive address space switches needed by traditional OSes[35].

Provided that the afore-mentioned trusted base contains a LTO-capable dynamic linker, using CGF would allow to blur the boundary between code running in different processes and even between user- and kernel-code. As an example, consider parameter or permission checking at the beginning of a system call from user to kernel code: if the compiler is able to prove by static analysis that the parameters will be valid or that the code has enough privileges, the validation could be skipped entirely. IPC could similarly benefit from CGF, eventually reaching the point where sending a message to a different "process" consists of atomically appending a pointer to a linked list.

Another intriguing possibility empowered by such a system would be specializing the code based on certain kernel conditions: as an example, let's consider some sort of cooperative multi-threading where the thread scheduler might guarantee that two pieces of code are never run concurrently. In this case all the relevant atomic/synchronization

operations might be either removed or converted to their non-atomic (and less expensive) variants.

2.3.4 Inter-procedural block motion

Based on trace data we can perform optimal block motion to minimize branching. As an example, consider the following code snippet:

```
1  function A() { // called Na times
2    // ...
3    C();
4    // ...
5  }
6
7  function B() { // called Nb << Na times
8    // ...
9    C();
10   // ...
11 }
```

In this case, if we assume that C is called much more often from A than from other functions (e.g. B), it makes sense to optimize the common case by putting the body of function B inside function A, while still allowing C to be called from B.

```
1  dispatch(func, param1) {
2    jump func;
3
4    dispatch_A:
5
6    A:
7      // body of function A
8      after_C = A_1;
9      // no jump here!
10
11   C:
12     // body of function C
13     goto after_C;
14
15   A_1:
16     // body of function A
17     return;
18
19   dispatch_B:
20
21   B:
```

```
22      // body of function B
23      after_C = B_1;
24      goto C;
25
26    B_1:
27      // body of function B
28      return;
29
30    dispatch_C:
31      after_C = C_1;
32      goto C;
33
34    C_1:
35      return;
36  }
```

We could obviously simply inline *C* in both *A* and *B*, but this leads to code duplication. CGF on the contrary allows us to rearrange the basic blocks and even getting rid of some jumps while avoiding code duplication.

Being able to perform inter-procedural code motion also allows the compiler to delay computations and create fail-fast paths, i.e. moving all the sanity checks before the calculations that depend on them. Consider this example[15]:

```
1  float sum_of_square_roots(float a, float b) {
2    return safe_sqrt(a) + safe_sqrt(b);
3  }
4
5  float safe_sqrt(float a) {
6    return a < 0 ? NaN : sqrt(a);
7  }
```

The compiler knows than any number when summed to a *NaN* yields *NaN*. This in turn means that if any one of the two numbers is negative, there's no point in calculating the square root of either one. In pseudo-code:

```
1  float sum_of_square_roots(float a, float b) {
2    return a < 0 || b < 0 ? NaN : sqrt(a) + sqrt(b);
3  }
```

Another powerful possibility is the ability to better perform instruction scheduling by allowing it to be done across function boundaries. For example, if a function generates high register or memory pressure it might be beneficial to interleave the function body with code from the caller(s) or the callee(s).

[15] on modern CPUs implementing the IEEE 754 standard for floating-point arithmetic the FPU already returns NaN in case the value is negative, so the check is not needed; on CPUs lacking a FPU, on the other hand, this functionality has to be implemented via software

2.3.5 Common Subexpression Elimination and Macro Compression

CSE is a compiler optimization that searches for instances of identical expressions (i.e., expression that all evaluate to the same value), and analyses whether it is worthwhile replacing them with a single variable holding the computed value. Let's consider the following simple example:

```
1 a = b * c + d;
2 e = b * c * f;
```

It may be worth transforming the code so that it is translated as if it had been written:

```
1 T1 = b * c;
2 a = T1 + d;
3 e = T1 * f;
```

The cost/benefit analysis performed by an optimizer will calculate whether the cost of the store to the temporary variable is less than the cost of the multiplication; in practice other factors such as which values are held in which registers are also significant.

In simple cases like this, programmers may manually eliminate the duplicate expressions while writing the code. The greatest source of common subexpressions are intermediate code sequences generated by the compiler, such as for array indexing calculations, where it is not possible for the developer to manually intervene. In some cases language features may create many duplicate expressions. For instance, C macros, where macro expansions may result in common subexpressions not apparent in the original source code.

The benefits of performing CSE are great enough that it is a commonly used optimization. Compilers need to be judicious about the number of temporaries created to hold values; an excessive number of temporary values creates register pressure possibly resulting in spilling registers to memory, which may take longer than simply recomputing an arithmetic result when it is needed[36].

If we extend CSE to work on sequence of instructions instead of simply subexpression, we have a powerful way to identify broader code repetitions. This extended version of CSE is known as macro compression[37].

2.3.5.1 Inter-procedural and inter-template

CGF allows CSE and macro compression to operate across function borders. While this might be not very powerful for CSE, macro compression could bring important code size reductions. Basically, this allows to find duplicated (or even just similar) computations in different functions.

In this case, interprocedural macro compression can be thought as automatically creating functions with code that is duplicated in different functions. This might help especially in languages with preprocessors that can generate lots of identical code (e.g. C macros). Intuitively this might mean turning C macros in functions, therefore reducing code size. The problem of determining an optimal set of macros that minimizes the space

required by a given code segment is known to be NP-complete, but efficient heuristics attain near-optimal results[38].

As a special case of interprocedural macro compression, we can also envision it being used to merge code across different functions. This may especially useful for templated code generated (common in certain languages, e.g. C++) by merging common sections shared by specialized versions of the same function.

```
1 template<class T>
2 bool function dump(vector<T> &v, FILE *fp) {
3   for (vector<T>::iterator i = v.begin(); i != v.end(); i++)
4     if (fwrite(&*i, 1, sizeof(T), fp) != sizeof(T))
5       return false;
6   return true;
7 }
```

In this example, much of the body of the function is the same independent of type T. Applying macro compression on such code after CGF should be able to factor out the common parts, effectively yielding a generic compiled function that can be called for all the appropriate values of type T. This should prove particularly effective in the heavily-templated code of the C++ STL (e.g. iterators).

Additionally this could be used also for functions that can return different data types.

2.3.5.2 Reverse constant propagation

When working with libraries translating/adapting idioms are quite common. For example, these could be of the form:

```
1  library_t translate_internal_to_library(internal_t c) {
2    switch (c) {
3      case INTERNAL_CONSTANT_A:
4        return LIBRARY_CONSTANT_A;
5      case INTERNAL_CONSTANT_B:
6      case INTERNAL_CONSTANT_C:
7        return LIBRARY_CONSTANT_B;
8    }
9    return LIBRARY_CONSTANT_C;
10 }
11
12 void internal_function(internal_t c) {
13   library_function(translate_internal_to_library(c));
14 }
15
16 void some_function(boolean b) {
17   if (b)
18     internal_function(INTERNAL_CONSTANT_A);
```

```
19    else
20      internal_function(INTERNAL_CONSTANT_B);
21  }
```

Performing full inlining and constant propagation on such code would obviously yield the following optimal version where the constants (both explicit constants and functions) have been propagated in reverse from the callee to the caller:

```
1  void some_function(boolean b) {
2    if (b)
3      library_function(LIBRARY_CONSTANT_A);
4    else
5      library_function(LIBRARY_CONSTANT_B);
6  }
```

The problem with full inlining have already been discussed, so it would be desirable to obtain the same effects without its drawbacks. CGF should allow this quite easily.

2.3.5.3 Other uses

Another use case might be, for example, merging different versions of the same code. This could be useful for big libraries/frameworks that ship multiple versions to support legacy code. This does not only mean that the size of the binaries is smaller, but also that a single binary has to be loaded in memory even if different versions of the framework are in use (this in turn means better cache performance). To a certain extent, this could be even used to pack together different versions of the OS itself, to avoid wasting space for multiple full copies. This is obviously not limited only to different versions of the OS, but also to builds with different build-time configurations.

2.3.6 Non-linear control flow

Non-linear control flow is a term used to denote when control flow deviates from its static behavior. This can be due to a number of causes, chiefly interrupts, stack manipulation, exception handling, coroutines or other non-linear constructs.

2.3.6.1 Interrupts

Interrupts are a mechanism that CPUs use to respond to (generally asynchronous) events. Interrupts are normally classified either as soft interrupts or hard interrupts. The latter are generated by devices attached to the CPU to signal some kind of asynchronous[16] event that should be handled[17]. The former on the other hand are interrupts generated by the currently-executing code, either explicitly using some special instruction (this is a widely

[16] in the sense that is not synchronous to the currently-executing code

[17] examples include the reception of a packet by a network interface card or the firing of a watchdog

used way of performing system calls) or as a consequence of some other instruction (page faults, illegal parameters, illegal instruction, etc.).

When an interrupt is raised, a minimal subset of the program state is automatically saved by the CPU and then the control is passed to an interrupt handler that will, eventually, save more program state to allow the proper interrupt processing to take place. The afore-mentioned minimal subset of program state will be automatically restored by the CPU when returning from the interrupt handler. It is important to note that generally the interrupt handler might be run in a different privilege level[18] (e.g. kernel level) from the code that was being executed when the interrupt was raised (e.g. user level).

If, on the contrary, this level switch is not needed (because the soft interrupt can be handled at the same privilege level), CGF could provide an attractive replacement for soft interrupts. Similar considerations apply for the non-linear control flow allowed by the *setjmp*/*longjmp* C functions that we discussed in section 2.2.3. CGF allows to statically keep track of which variables are in use at each point of the program, so it can be used to semantically replace these two functions.

2.3.6.2 Coroutines

Coroutines generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations. When subroutines are invoked, execution begins at the beginning and once a subroutine exits, it is finished; an instance of a subroutine only returns once. Coroutines are similar except they can also exit by yielding to other coroutines, which allows them to be re-entered at that point again; from the coroutine's point of view, it is not exiting at all but simply calling another coroutine.

Coroutines are well-suited for implementing more familiar program components such as cooperative tasks (e.g. producer-consumer), iterators, infinite lists and pipes. Consider for example the *fibonacci* generator function below that generates the part of the Fibonacci sequence. Basically the execution starts from the beginning of the function and yields on line 5. At each successive invocation of the function execution will resume from line 6 and yield on line 5 or terminate on line 10.

```
1  function fibonacci() {
2    a = 0;
3    b = 1;
4    do {
5      yield b;
6      t = a;
7      a = b;
8      b = t + a;
9    } while (a < 1000);
10   return;
11 }
```

[18] also known as CPU ring level

As far as CGF is concerned, coroutines are not different from the interrupts described in the previous section: "yielding to" in this context means basically raising a soft-interrupt that gets handled by the other coroutine. This could allow a very efficient implementation of coroutines and many other non-linear control flow constructs. Among these exception handling is probably the most used and will be discussed in the next section.

2.3.6.3 Exception handling (EH)

An interesting application of CGF is almost-zero-cost exception handling without runtime support.

EH is normally implemented either using soft interrupts or using error codes.

Interrupt-based EH The most commonly used one is generating a software interrupt when a throw is executed. This software interrupt saves the state of the thread of execution (stack, registers, etc.) and transfers control to a personality function that inspects this state, decides which catch block will receive control, performs stack unwinding as needed (eventually destroying the objects on the stack), changes the instruction pointer to point to the catch block and terminates. When the normal flow of execution restarts, the catch block is executed.

This system ensures that the performance of the normal flow of execution is not hindered by the exception handling code, because all of it (supposedly rarely executed) is placed in a distant section of the code (hence the name of zero-cost exception handling, where the cost is supposedly considered only for normal execution). All of this obviously makes exception handling rather costly in terms of execution speed (when the exception is actually executed), in terms of compiler support (in the form of runtime support) and in terms of interoperability (mixing exception handling code generated by different compilers is generally unreliable).

Error-code-based EH The other way is the C way of providing exception handling, i.e. returning special values from the called function to signal an error condition.

This is obviously more portable - because in general calling conventions are respected by all compilers - does not rely on runtime support (in the form of software interrupts, interrupt handlers, etc.), is faster in the exception case but is slower in the normal case because function call overhead is present on the normal flow of execution. Additionally, another problem with this approach is that the programmer is generally required to write a lot of code to deal with error conditions, and this either means writing fragile code or using abstractions that further hurt performance.

CGF on the other hand could be used to solve the problem rather elegantly by providing almost-zero cost EH in the caller without requiring runtime support or non-portable solutions. If we additionally support transforming catches and throws, we could exploit the same mechanism to flatten exception handling. By "almost-zero cost" we mean that there is a small runtime overhead due to it, but it should be no higher than pushing

a single pointer on the stack for each function containing a try block. The expected performance impact should therefore be negligible (if not downright zero) in most cases.

As an example, let's consider the following code snippet (function *D* is omitted for the sake of brevity):

```
1  A() {
2    try {
3      B();
4    } catch {
5      D();
6    }
7  }
8
9  B() {
10   // ...
11   C();
12   // ...
13 }
14
15 C() {
16   // ...
17   if (error_condition)
18     throw;
19 }
```

if an exception were to be thrown inside *C*, the control should be passed to the catch block in *A* and then to *D*. By performing CGF, the compiler can perform optimizations that lead to actually pass the control directly to *D* as soon as an exception is thrown in *C*:

```
1  dispatch(func) {
2    dispatch_start:
3      goto func;
4
5    dispatch_A:
6      after_A = dispatch_end;
7      catch_A = dispatch_uncaught;
8    A:
9      catch_A = A_2;
10     after_B = A_1;
11     catch_B = catch_A;
12     goto B;
13
14   A_1:
15     catch_A = dispatch_uncaught;
```

```
16      goto after_A;
17
18    A_2:
19      catch_A = dispatch_uncaught;
20      after_D = A_1;
21      catch_D = catch_A;
22      goto D;
23
24    dispatch_B:
25      after_B = dispatch_end;
26      catch_B = dispatch_uncaught;
27    B:
28      // ...
29      after_C = B_1;
30      catch_C = catch_B;
31      goto C;
32
33    B_1:
34      // ...
35      goto after_B;
36
37    dispatch_C:
38      after_C = dispatch_end;
39      catch_C = dispatch_uncaught;
40    C:
41      // ...
42      goto error_condition ? catch_C : after_C;
43
44    dispatch_D:
45      after_D = dispatch_end;
46      catch_D = dispatch_uncaught;
47    D:
48      // ...
49      goto after_D;
50
51    dispatch_end:
52      return;
53
54    dispatch_uncaught:
55    // uncaught exception!
56  }
```

as usual, if we assume that the common case is calling A, we can optimize away the rather verbose constructs generated by CGF to get:

```
1  dispatch(func) {
2    dispatch_start:
3      goto func;
4
5    dispatch_A:
6    A:
7      after_B = A_1;
8      catch_B = D;
9    B:
10     // ...
11     after_C = B_1;
12     catch_C = catch_B;
13   C:
14     goto error_condition ? catch_C : after_C;
15
16   B_1:
17     // ...
18     goto after_B;
19
20   A_1:
21     return;
22
23   A_2:
24   D:
25     // ...
26     return;
27
28   dispatch_B:
29     after_B = dispatch_end;
30     catch_B = dispatch_uncaught;
31     goto B;
32
33   dispatch_C:
34     after_C = dispatch_end;
35     catch_C = dispatch_uncaught;
36     goto C;
37
38   dispatch_D:
39     after_D = dispatch_end;
40     catch_D = dispatch_uncaught;
41     goto D;
42
```

```
43    dispatch_end:
44      return;
45
46    dispatch_uncaught:
47    // uncaught exception!
48  }
```

Normally if C were to throw, we would have to perform two returns and one call ($C \rightarrow B \rightarrow A \rightarrow D$). After CGF the throw is transformed in a single jump to the catch site in A. Since the catch site only contains an unconditional jump to D, the whole throw can be optimized to a jump to D.

This could be easily extended to accommodate conditional rethrows, eventually providing the foundation for almost-zero-cost SEH.

It is important to note that such a transformation would completely remove the distinction between EH-aware and EH-unaware code in the IR. As such, optimizations should not be hindered anymore by platform-dependent EH semantics.

2.4 Obstacles

CGF, as presented until now, has a number of issues that should be resolved to be effective on real-world code. In this chapter we present the most important ones along with a number of considerations that should help to fix them.

As a rule of thumb, CGF suffers from the same class of problems that affect inlining (inlining hazards). The authors of [20] identify some conditions that hinder successful flattening: indirect calls, explicit modifications of the stack pointer and recursion. In the following sections we expand their analysis.

If none of these hazards are present then the whole program can be flattened in the *main()* function, no wrappers are needed and it is possible to avoid using the stack completely (see section 2.5.2)

2.4.1 Recursion

Recursion is the prototypical inlining hazard. Traditionally recursion happens when the same function F is present more than once in the call stack.

There are two types of recursion, direct and indirect. The former happens when a function F calls itself directly. This type of recursion is rather easy to detect because it's enough to scan the body of the function for calls to the function itself. The latter, on the other hand, happens when F calls another function that either calls F or calls another function that will ultimately lead to a call to F. To detect such kind of recursion, the call graph must be analyzed to identify cycles.

The reason why recursion is not suited to flattening is because CGF has to statically keep track of the live variables in the caller(s) during the execution of the callee. If a function is recursive CGF needs to keep (exactly as it happens on the stack in the case

of regular recursion) an additional copy of the live variables for each recursion level. This is not feasible if the depth of the recursion must be assumed unbounded at compile time (this would require keeping track of infinite live variables) but can be done if either we know an upper bound to the recursion depth or if we simply set an arbitrary hard limit (in this case if the program attempts to go past the hard limit it would trigger a stack overflow similar to what is done at runtime by the OS).

There are two additional conditions that should be considered as well. The first one is that if the recursive call is in tail position (i.e. the call is the last instruction), there are no live variables to be tracked[19] and therefore the above limitation does not apply. The recursion is effectively transformed in a simple loop.

The second condition happens when analysis is able to prove that two live variables from different recursion depths actually contain the same value: in this case duplicates should not get stored on the stack.

As a side node, it is obviously beneficial to recall that it is not compulsory for callsites in the dispatch function to be flattened: they could be very well left as-is provided that either the callee has been flattened and the relative wrapper has been created or the callee has not been flattened at all.

2.4.2 Indirect function calls

Flattening the CFG is possible only when the CFG itself is statically known. In particular, a callsite can not be flattened if it can not be proven to call only functions transformed by CGF.

A possible solution to this can be to insert a check before the jump: if the address to jump to does not belong to the current function then it should perform a regular call.

```
1  switch (call_target) {
2    case func_1:
3      after_cgf_func_1 = after_call;
4      goto cgf_func_1;
5    case func_2:
6      after_cgf_func_2 = after_call;
7      goto cgf_func_2;
8    default:
9      call_target();
10 }
11 after_call:
12   // ...
```

If call_target can be proven to be a single function by constant propagation or data-flow analysis, then we can turn the indirect callsite into a direct callsite (and therefore into a unconditional branch). If call_target is proven to be more than one function but all

[19] actually it is more accurate to say that the converse is true, i.e. a call is in tail position if there are no live variables at the callsite

of them are going to be CGF-transformed then we can turn the callsite into an indirect branch. Otherwise we have to generate a regular indirect call (possibly towards one of the wrappers).

2.4.3 Stack manipulation

Since the stack is not used (or used in non-standard ways) by CGF-transformed code, functions that explicitly operate on the stack, stack pointer or instruction counter will probably not work or crash the program.

Some of them, like *setjmp*/*longjmp*, can be ported easily because they provide non-local control flow semantics that can be natively modeled in CGF.

2.4.4 Function signatures

First of all, it is necessary to properly scope the problem of differing function signatures, i.e. different parameter types, number of parameters, return value types, number of return values and function attributes/modifiers (including calling conventions).

This problem arises because after CGF all the original functions are merged in the Σ-function. If any of these functions are to be called from external code, we need a way to forward the call to the original function (with its signature) to the corresponding function embedded in Σ.

As such, this is a problem if and only if we need to perform a call across the boundary between normal and CGF-transformed code.

Luckily, there is a simple solution to this problem, i.e. creating forwarding wrappers with the signature of the original function that

1. wrap the incoming parameters in an opaque structure to be passed to the Σ-function
2. allocate the space for the return values
3. call the corresponding function embedded in the Σ-function, passing the opaque structure and an opaque pointer to where to store the return values
4. returns the return values

Consider as an example the following two functions:

```
1 int A(int a, void* ptr) {
2   // ...
3 }
4
5 float B(double b) {
6   // ...
7 }
```

the corresponding forwarding wrappers would be:

```
1  #define OPAQUE(x) ((void*)(&(x)))
2
3  struct param_A {
4    int a;
5    void *ptr;
6  }
7
8  struct retval_A {
9    int retval;
10 }
11
12 int A(int a, void *ptr) {
13   struct param_A param = { a, ptr };
14   struct retval_A retval;
15   dispatch(dispatch_A, OPAQUE(param), OPAQUE(retval));
16   return retval.retval;
17 }
18
19 struct param_B {
20   double b;
21 }
22
23 struct retval_B {
24   float retval;
25 }
26
27 float B(double b) {
28   struct param_B param = { b };
29   struct retval_B retval;
30   dispatch(dispatch_B, OPAQUE(param), OPAQUE(retval));
31   return retval.retval;
32 }
```

afterwards, in the corresponding dispatch landing pad, the opaque pointers are casted back to their respective types:

```
1  dispatch(func, param, retval) {
2    dispatch_start:
3      goto func;
4
5    dispatch_A:
6      int A_a = ((param_A*)param)->a;
7      void *A_ptr = ((param_A*)param)->ptr;
8      // ...
```

```
 9
10    dispatch_A_end:
11      ((retval_A*)retval)->retval = (int)retval_A;
12      return;
13
14    dispatch_B:
15      double B_b = ((param_B*)param)->b;
16      // ...
17
18    dispatch_A_end:
19      ((retval_B*)retval)->retval = (float)retval_A;
20      return;
21  }
```

A somewhat related problem is posed by variadic functions[20], i.e. functions that accept a varying number of parameters by directly accessing the memory locations on the stack. For such functions, we have a number of behaviours we can envision: the easiest is to simply refuse to perform CGF on variadic functions but others are possible as well (such as forwarding a pointer to the variadic parameters to the Σ-function or even specializing a version of the function for each variadic combination and then having macro compression merge them together).

2.4.5 Interactions with other transformations

It is reasonable to expect CGF to interact poorly with other optimizations if not done in the correct order. Roughly speaking, it can be said that simple intra-procedural optimizations can be safely run before CGF whereas all other optimizations should be run afterward.

2.4.5.1 Prerequisites

CGF is likely to benefit if some optimizations are done before it. In this section we highlight some of the transformations that should be done before CGF is applied. While no one of these is strictly needed, using them should result in reduced work for subsequent optimizations - therefore lowering compilation time.

Value promotion Many languages have various semantics that must be translated in the IR for the program to be correct. An example of this is the requirement in C for all local functions to be pointer-addressable. Value promotion deals with many spurious results of such semantics and yields a much leaner IR to work on.

Callsite canonicalization This is a pre-pass that ensures that all call instructions are the last instruction in their basic block, so that when flattening the callsite we have a basic block where we can jump back from the callee.

[20] also known as vararg functions, well-known examples include printf() and scanf()

This means that something like this:

```
function A() {
  // ...
  B();
  // ...
}
```

is turned into:

```
function A() {
  // ...
  B();
  goto A_1; // this goto will be unreacheable after flatten-
            // ing because at the end of B() we'll jump to
            // A_1 we insert it because SSA requires all BBs
            // to be terminated by a terminal instruction

  A_1:
  // ...
}
```

Return instruction unification This pass is not strictly needed but simplifies the flattening phase because in this way we need to keep track of a single return instruction.

This means that something like this:

```
function randomSign() {
  if (rand() % 2)
    return 1;
  return -1;
}
```

is turned into:

```
function randomSign() {
  if (rand() % 2) {
    retval = 1;
    goto ret_randomSign;
  }
  retval = -1;
  goto ret_randomSign;

  ret_randomSign:
    return retval;
}
```

Later optimizations, after CGF has run, will split back the jump to the caller if it is profitable to do so.

Constant propagation CGF can work well only if the call graph is statically known. If indirect callsites are present we can't easily transform them.

In this cases, constant propagation might help because it may allow us to restrict the number of functions that might be called by each callsite. If it can prove that a certain indirect callsite will always call the same function, we can simply transform the indirect callsite to a direct one. Similarly, if it finds more than one possible callee we can provide static branches to them.

Consider, for example, the following snippet:

```
1  (int)(*F)();
2
3  int main() {
4    F = &foo;
5    return bar();
6  }
7
8  int foo() {
9    return 0;
10 }
11
12 int bar() {
13   return F();
14 }
```

the call to *F* in *bar* is an indirect call. By performing constant propagation we should find out that *F* is only assigned one single value (*&foo*) in the whole program and therefore the indirect call *F* is simply a direct call to *foo*[21].

Basic inlining If a function is called only from a single callsite and is not available externally it is always profitable to inline it. Doing it before CGF (and removing the resulting dead function) will simply speed up the transformation itself by lowering the number of functions to work on.

2.4.5.2 Postrequisites

After CGF has run all kinds of optimizations are supposed to be run. In this section we highlight the ones that should be able to provide the greatest improvements.

[21] this would obviously not hold if F had been marked volatile

CSE and macro compression Common-subexpression elimination and macro compression are two transformations that scan the the program for duplicated code. The former operates typically on sub-parts of a single expression whereas the latter operates on sequences of instructions.

When such sequences are found, they are either merged together (to reduce cache pollution) or their result is saved to be reused (to avoid duplicated computations).

Normally such transformations are applied to a single function at a time, so duplicated instructions in different functions are not optimized. CGF, by merging all the code in a single function, allows working on code from different functions. See section 2.3.5 for more details.

Invariant code motion and other loop transformations Invariant code motion is a loop transformation that moves out of the loop calculations that do not change throughout the loop. CGF makes it possible to perform this optimization across function boundaries.

Similar considerations can be done for all the common loop transformations.

Jump threading Jump threading is a pass that removes unnecessary conditional branches by merging the respective blocks as needed. The example in section 2.3.3.2 is an application of jump threading.

CGF is particularly important for this optimization because it makes it possible to thread together blocks from different functions. Coupled together, they make possible to support almost-zero-cost non-linear control flows.

Vectorization and parallelization Code vectorization is a family of transformations that enables sequential code to be executed by SIMD units commonly found on modern processors or by other special-purpose processors (GPUs, FPGAs, etc.).

Thanks to the enhanced ability to perform code motion across function boundaries such transformations could become much more effective: data dependencies that might hinder vectorization can be broken early by loop splitting or code motion[39, 7].

2.4.6 Compile-time issues

Code transformed by CGF might pose a number of problems to other optimization passes and to the backend because it significantly increase the complexity of the code undergoing compilation. This is mainly due to the proliferation of predecessors for many basic blocks and to the great amount of ϕ nodes inserted to keep track of the live variables.

2.4.6.1 Branching complexity

Normally calling conventions provide a standard that each function must conform to, either when calling another function or when being called. This greatly simplifies passing the parameters to and from each function but also introduces overhead[15].

In CGF, most overheads are removed but at the expense of having to define the optimal calling convention for each callsite. Intuitively this can be attributed to the fact that we must forward to the callee all live variables at the callsite in the caller. For example, if a function F is called from M different callsites C_i each having V_i live variables, the compiler will have to track $\sum_{i=1}^{M} V_i$ live variables through F alone[22]. Moreover, these live variables will be inherited by any callee as well and, due to the characteristics of the SSA representation, will have to be explicitely tracked in the union of the postdominance frontier of all callsites in the function. This means that the number of phi nodes needed to track the live variables is expected to be $O(N^2)$, where N is the number of callsites in the program.

While this allows a great deal of flexibility for the compiler to perform optimizations that would not be otherwise possible, it is also the main cause of the code size increase. This could be partially resolved by using some adaptivity when it comes to subsequent optimizations, e.g. favoring code speed over code size when working on frequently executed BBs and the contrary when working on rarely executed BBs (this would obviously be more significant as a PGO).

2.4.6.2 Register allocation

CGF is likely to increase the load on the register allocator because register allocation has to be applied globally to all the code in the module instead of separately to each function.

It should be noted that to make register allocation really effective should be made PGO-aware so that register allocations across frequently-executed branches is given priority over rarely-executed branches.

2.4.6.3 Stack management

Normally, all local variables in the current function are pushed to the stack when a call is executed. While this is usually not a problem in regular functions, it might be a problem if all the code in the module (along with all its local variables) is merged in a single function, because a single stack frame must contain all the variables in the module.

It's therefore imperative that either normal calls to function are avoided (this includes recursive calls to the CGF function) or, better yet, that the compiler is able to accurately compute the set of live variables across each of the external function calls and that register spilling is implemented in a way than enables reusing dead locations in the stack frame of the global function.

If this is not done properly, in complex programs stack fragmentation could start causing excessive stack usage. To see why this could be the case, let's suppose we are running a CGF-transformed program. One of the ancestors might have had to push a variable on the stack because it wasn't going to be used soon. After a few call layers we are now in a descendant of that function and we need the value held by that variable.

[22] note that these live variables are formally unused inside F: they will be simply passed back to the caller when jumping back to it

We read it from the stack and now the variable is dead. Shortly afterwards, we need to store a new variable on the stack (let's assume because of register pressure). We are now facing a big problem: where do we put it?

A naive implementation would simply push the new variable on top of the stack. But this doesn't take into account the fact that there is a dead variable in the stack and we could reuse thst location for the new variable.

A simple implementation would probably just scan (statically at compile-time) the stack for a suitable reusable location and store the value there. The problem with this is that it may happen that a variable of size S_V is placed in a reusable location of size $S_L < S_V$, leaving an unused (and hardly usable) location of size $S_L - S_V$. With time this small fragments might add up and make the stack grow.

The right way of doing this would obviously be statically determinating the best layout of data[23] to make sure that no more than the space that is strictly needed is used.

This problem has many similarities to that of register allocation, so theoretical and heuristic solutions are well-known.

2.4.6.4 Debugging

All optimizations have the tendency to disrupt the possibility of meaningful debugging by making it very hard to associate lines and variables present in the source code to instructions and registers/memory locations in object code[40].

CGF in this respect would require changes to the debuggers because all functions would run in the same stack frame: as such the debugger would need to rely on the static analysis and extended mapping information provided by the compiler.

2.4.7 Runtime issues

Besides the increased burden on the compiler, CGF-transformed code might also cause some issues at runtime due to the unusual code structure.

2.4.7.1 CPU pipeline

Indirect and conditional branching are well known causes of pipeline stalls[24] on modern CPUs.

A pipeline stall happens when an instruction has to wait for one or more preceding instructions to complete. This is normally due to the fact that the current instruction uses as input the output of the preceding instructions or, more generally, the behavior of the current instruction depends on the previous ones.

In the context of indirect branching, the problem is that the jump target itself is dependent on the result of the previous instructions. This means that either the CPU waits for the previous instructions to complete before loading the code at the jump target and jumping there or that the CPU speculatively predicts which one will be the likely

[23] this might also mean having to perform stack defragmentation
[24] also known as pipeline bubbles

jump target and starts fetching and executing it. If the prediction turns out to be correct, the CPU has already started executing the branch. On the other hand, if the prediction was wrong then the CPU has to undo any calculations that were done as a result of the wrong prediction, flush the pipeline from outstanding instructions from the predicted branch, fetch the correct branch and start executing it. On modern x86 processors the penalty for a mispredicted branch goes from 10 to 30 CPU cycles[10].

Speculative execution is therefore a double-edged sword: when the branch predictor is correct the CPU saves a lot of time, but if the branch predictor does a mistake the execution is actually slower than having no branch prediction at all.

The problem with CGF is that potentially it inserts lots of indirect branches, especially when returning from the callee. It must be noted though that these indirect branches were already there in the first place in the form of the return instruction[25] and that most of them should depend on values[26] that have been set well ahead of time[27], so the branch predictor should be able to correctly predict most of them.

2.4.7.2 Jump target alignment

On many CPU architectures it is beneficial that jump targets (i.e. instructions that may be the target of a jump) are aligned to certain boundaries.

Most microprocessors indeed fetch code in aligned 16-byte or 32-byte blocks. If an important subroutine entry or jump label happens to be near the end of a 16-byte block then the microprocessor will only get a few useful bytes of code when fetching that block of code. It may have to fetch the next 16 bytes too before it can decode the first instructions after the label. This can be avoided by aligning important subroutine entries and loop entries by 16. Aligning by 8 will assure that at least 8 bytes of code can be loaded with the first instruction fetch, which may be sufficient if the instructions are small. We may align subroutine entries by the cache line size (typically 64 bytes) if the subroutine is part of a critical hot spot and the preceding code is unlikely to be executed in the same context. The main disadvantage of code alignment is that the empty space before the jump target must be filled with NOPs: this increases code size and makes code caching less efficient[41].

On the other hand, caching of code works best if critical code paths are contained within a contiguous area of memory no bigger than the code cache. Simplifying a bit, we could say that code that is often executed together should be as closely packed as possible. Rarely accessed code such as error handling routines should be kept instead separate from the critical hot spot code[42].

Fortunately, modern compilers already do these operations when optimizations are enabled.

[25] even though in many CPU architectures the call/return mechanism is highly optimized with dedicated return address buffers and predictors, and therefore replacing call/return with jumps will not benefit from them anymore

[26] that will likely be held in registers

[27] in the caller

At the same time it should be noted that CGF *per se* does not create new branches - it merely turns call and return instruction in jumps. And since function alignment is probably more strict than jump alignment, CGF should not worsen the situation.

2.5 Side effects

A number of interesting side effects might result from the use of CGF. In this section we list the prominent ones, mainly because they open up possible new research directions.

2.5.1 Code obfuscation

The most obvious side effect is code obfuscation because the (relatively) high-level subdivision of the program in functions is completely lost.

While this *per se* does note really provide much obfuscation (all the code is still there, mostly in the same form as before) subsequent optimizations will likely perform extensive changes, finally unhindered by function boundaries, possibly allowing code from different functions to be mixed together[28].

Indeed real code obfuscation techniques operating on the CFG basically rely on modifying the CFG so that all (or most) basic blocks appear to have the same set of predecessors and successors[17]. The actual control-flow during execution is then determined dynamically by a dispatcher, generally by exploiting opaque predicates, i.e. properties of the program known by the compiler that are hard or impossible to statically infer from the compiled code (e.g. opaquely non-deterministic predicates[19], aliased opaque predicates[16], dynamically opaque predicates[18]). Since our target is not code obfuscation this techniques are not applied.

2.5.2 Stackless execution

In [20] the authors highlight the fact that if the code being compiled is completely hazard-free (see section 2.4) the whole program can be flattened to a single function (the traditional *main()* function) and that an upper bound on the growth of the stack is statically known.

This means that there is no need for dedicated stack support in the CPU and OS and that registers normally used to track the stack can be theoretically turned into general-purpose registers.

2.5.3 Protection against stack smashing

Stack smashing[29] is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory on the stack. This is a special case of violation of memory safety.

[28] this effect is also known as code interleaving[16]

[29] also known as stack buffer overflow

Buffer overflows can be triggered by inputs that are designed to execute code, or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security. They are thus the basis of many software vulnerabilities and can be maliciously exploited to manipulate the program in one of several ways:

1. By overwriting a local variable that is near the buffer in memory on the stack to change the behaviour of the program which may benefit the attacker.
2. By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user input filled buffer[30].
3. By overwriting a function pointer, or exception handler, which is subsequently executed.

Because in CGF all functions have been merged and because we don't rely on known calling conventions to transfer control to other functions, CGF makes it quite hard to exploit stack smashing to alter the normal flow of the program.

Indeed, most live values (possibly including the return address of the current function) are kept in registers or even out-of-band (i.e. metadata). Moreover, even if it may be possible for some of them to be pushed to the stack, they are pushed in a not easily predictable order, possibly dependent on the previous callers. This especially hinders calling arbitrary functions, because the attacker should know the exact registers and stack positions where the function expects its inputs (and they will likely be unique to each callsite). In [20] this property of CGF is proved, albeit on a slightly different implementation that guarantees that pointers used for dispatching are never stored on the stack.

As a further degree of protection, the register allocator and stack spiller could be modified to make non-deterministic allocation and spilling choices: in this case, two builds of the same source code would be radically different in how parameters are passed between functions. If this step were to be done at runtime there would probably be no easy way to reliably exploit stack buffer overflows.

2.5.4 Code size increase

In [20] the authors note the tendency of CGF to increase the code size of the compiled program, even if they acknowledge that selective flattening might help to mitigate this effect.

Another way to mitigate this effect would be implementing a robust and adaptive macro compression transform that would aggressively merge rarely-executed segments. Furthermore it could be interesting devising some kind of automatic code threading[31] to further increase code density: this could be very useful for large loops containing many calls.

[30] this attack is also known as return-to-libc attack[43]

[31] see section 2.2.4 for a description of threaded code

3 Implementation

In this section we present the algorithm that performs CGF and the details of the implementation as an LLVM transformation.

To describe the algorithm it is beneficial to define ϕ_{dummy}, a modified version of the regular ϕ node. Whereas regular phi nodes need to explicitly list all possible incoming paths and their associated values of the ϕ, as we saw in section 1.3.3.3, ϕ_{dummy} assumes a defined value only for a single incoming path[1]:

$$\phi_{dummy}(V_1, F_1) = \begin{cases} V_1 & \textit{if control comes from } F_1 \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

This is a useful shorthand notation because live values are unique to each caller and are not logically accessible from the callee or from other callers. Without this notation, we would have to write long sequences of:

$$\phi(V_1, F_1, \textit{undefined}, F_2, \textit{undefined}, F_3, ...\textit{undefined}, F_N)$$

It will be then up to the compiler to perform optimal phi nodes fusion, register allocation, spilling and rematerialization for every F_i.

3.1 High-level overview

CGF is a two-phases transformation operating on the whole module. In this section we describe the algorithms used to implement this transformation.

[1] more generally, ϕ_{dummy} could allow more than one incoming path with a defined value and a default value V_D for the other cases, i.e. ϕ_{dummy} would assume the value V_D only for incoming paths that have not been explicitly declared:

$$\phi_{dummy}(V_D, V_1, F_1, V_2, F_2, ..., V_N, F_N) = \begin{cases} V_1 & \textit{if control comes from } F_1 \\ V_2 & \textit{if control comes from } F_2 \\ ... & ... \\ V_N & \textit{if control comes from } F_N \\ V_D & \textit{otherwise} \end{cases}$$

Since this latter semantic is not needed in the present context we will use the former one.

Algorithm 3.1 Incoming live values enumeration algorithm (exact)

$D_B \leftarrow$ `set of dominators of basic block` B
$S_B \leftarrow$ `set containing b and its successors`
$VD_B \leftarrow$ `set of the values defined in` D_B
$VS_B \leftarrow$ `set of the values with uses in` S_B
$LV_B \leftarrow VD_B \cap VS_B$

Algorithm 3.2 Incoming live values enumeration algorithm (over-approximation)

$D_B \leftarrow$ `set of dominators of basic block` B
$VD_B \leftarrow$ `set of the values defined in` D_B
$LV_B \leftarrow VD_B$

3.1.1 Analysis phase

The first phase analyzes and prepares the code in the module for the actual transformation. This corresponds to the first loop of algorithm 3.3.

In particular it makes sure all call instructions are the last instruction in their basic block and that any following code is moved to a different basic block. This is done to have a basic block to jump to when returning from the call, after the transformation. This transformation has been discussed in detail in paragraph "Callsite canonicalization" of section 2.4.5.1.

It then proceeds to create a mapping between the callsite and a number of information that will be used in the proper transform. In particular it identifies the following:

- return basic block: address of the basic block created immediately after the callsite that will receive control when the function returns
- callsite basic block: address of the basic block containing the callsite (will be used for the phi nodes in the callee)
- set of values live across the callsite: these need to be preserved and forwarded to the return basic block (see algorithms 3.1 and 3.2 for examples of how to enumerate the values that are live at the beginning of a basic block)
- set of values passed to the callee as arguments
- set of values returned from the callee as return values (some functions might return more than one value)
- set of possible callees of the callsite (an indirect callsite might call more than one function)

The above are all the information we need to succesfully perform CGF. Once done we can move to the proper flattening phase.

3.1.2 Flattening phase

The second phase starts by creating Σ, the empty dispatch function that will contain the flattened code. We then proceed to flatten the functions and the callsites and to propagate the live values across function calls.

Flattening the functions basically means placing a copy of the body of all functions in the module inside the dispatch function while keeping track of the mapping between entities (values, BBs, instructions, etc.) in the original functions and in the dispatch function.

Flattening the callsites means replacing *call* and *return* instructions with *jump* instructions and fixing the arguments/return value/live values passing between the callee and caller.

Fixing the arguments is quite easy because it simply means adding an incoming value to the ϕ node in the callee that replaces the formal argument.

Fixing the return value is easy as well because it simply means getting a reference to the value that was to be returned by the *return* instruction in the callee and replacing with it the uses of the value returned by the *call* instruction in the caller.

Fixing the local live values is, on the other hand, somewhat more difficult because it requires replacing their uses only in the return BB and its successors. This can be done by using the set of successors S_B (see algorithm 3.1) of the callsite being flattened and iterating on the uses of the live values in LV_B: if a use is located in S_B the used value has to be replaced with the corresponding live value coming from the return BB of the callee.

3.2 Framework

To implement CGF the Low Level Virtual Machine[44] (LLVM) was chosen.

LLVM is a modular compiler infrastructure, written in C++, which is designed for compile-time, link-time, run-time, and idle-time optimization of programs written in arbitrary programming languages. The language-agnostic design of LLVM has allowed the creation of a wide variety of front ends, including C, C++, Objective-C, Fortran, Ada, Haskell, Java bytecode, Python, Ruby, ActionScript, GLSL, and others (e.g. GCC-IR).

LLVM can provide the middle layers of a complete compiler system by operating on the LLVM-IR, a language-independent instruction set and type system based on SSA form. This IR can then be converted and linked into machine-dependent assembly code for a target platform (either at compile-time or run-time).

Being a fairly new and active project, its syntax and semantics are still unfrozen and the documentation is not always up-to-date. Nevertheless its design is rather clean and its performance is quite good when compared to more mature projects like GCC. Benchmarks found that LLVM trails GCC in code quality (speed of the compiled programs) by about 10% on average, while compiling 20-30% faster[45].

Algorithm 3.3 Call-graph flattening algorithm

```
for each function F in the module
  unify the return instructions of F
  for each callsite S in function F
    BB_S ← split the basic block immediately after callsite S
    V_S ← live values across callsite S
    A_S ← arguments of callsite S
    R_S ← return value of callsite S
    C_S ← possible call targets of S
    P_S ← basic block containing callsite S
    callsites[S] ← {BB_S, V_S, A_S, R_S, C_S, P_S}
Σ ← create empty dispatch function
for each function F in the module
  clone the body of F in Σ
  E_F ← entry block of the cloned F
  X_F ← exit block of the cloned F
  R_F ← value returned by the return in X_F
  RA_F ← create an empty φ node in E_F
  replace the return in X_F with a jump to the value of RA_F
  for each callsite S in callsites
    {BB_S, V_S, A_S, R_S, C_S, P_S} ← callsites[S]
    if F is in C_S
      replace S with a jump to E_F
      replace all uses of R_S with R_F
      add BB_S to RA_F in E_F
      add each argument in A_S to the corresponding arg in E_F
      for each live value V in V_S
        P ← create a φ_dummy(V, P_S) in E_F
       replace all uses of V with P in BB_S and its descendants
```

3.3 Coding

The CGF transform has been implemented in C++ as a module transformation (ModulePass) in LLVM 2.8[2] running on Ubuntu 11.04 x64.

Module transformations are the most powerful transformations available in the LLVM framework because they can operate on the module as a whole (akin to interprocedural optimizations). This was required because CGF has to create the dispatch function and to move code between functions.

The current implementation makes a number of assumptions (described in section 3.3.1) that renders it not ready for real-world usage. Its primary aim was merely that of verifying that the transform could be used on processors (such as the x86 family) more complex than those targeted in [20] and to provide validation of some of the ideas proposed in this thesis.

The code was broken down in four functional classes. This was done to keep it tidy and well scoped. Overall, the code amounts to about 800 lines of C++.

CGFFunction Represents a function being transformed. Encapsulates a LLVM Function. Performs the flattening of the function body in the dispatch function and keeps track of external live values through the function.

CGFCallSite Represents a function call. Encapsulates a LLVM CallInst. Performs the transformation of the call instruction into a jump and keeps track of the local live values across the function call.

CGF Main class of the CGF transform. Enumerates the functions and callsites and flattens them.

CGFPass Boilerplate class implementing the LLVM ModulePass interface. Instantiates and invokes the main CGF class.

3.3.1 Assumptions

Typical sources of inlining hazards are not currently supported in our implementation. The following list include conditions that are not supported:

Indirect static calls Indirect static calls are calls to a function pointer whose possible values are known at compile-time. Support for indirect static calls can be easily added, it would merely require adding the appropriate ϕ nodes in the return BB for the return value and live values.

[2] revision 127244

Indirect dynamic calls Indirect dynamic calls are calls to a function pointer whose possible values are not (fully) known at compile-time. They can't be flattened as-is because we need to statically know the call graph.

Common source of such calls are dynamically-linked libraries or (more rarely) self-modifying code.

There are obviously ways around this: first of all the fact that they can't be flattened doesn't prevent their use - the call will be simply left as-is and won't benefit from other optimizations. If some (but not all) possible values are known at compile time we can flatten the call and provide a default branch that will be taken if an unknown pointer is called:

```
1 switch (funcPtr) {
2   case funcA: goto CGF_funcA;
3   case funcB: goto CGF_funcB;
4   // ...
5   default: funcPtr(); goto retBB;
6 }
```

Moreover, nothing prevents from running CGF (and the subsequent optimizations) at link-time or run-time: in this way the targets of many indirect dynamic calls would be known and therefore they could be treated as indirect static calls.

Recursive tail calls Recursive tail calls are recursive direct calls where the *call* instruction is in tail position. This means that they don't have any local live value and as such they can essentially be treated like a loop. As such, support for such calls can be easily added.

Recursive direct calls Recursive direct calls are call instructions that directly calls the function they belong to. A special case of such calls are recursive tail calls (discussed above). Recursive non-tail calls can not be flattened as-is because they contain local live values that need to be tracked (unless the live values don't change during recursion).

As in the case of other unsupported conditions, the fact that in our pilot implementation they can not be flattened does not mean that they can not be used: the call will be left as-is and the original function will simply forward the call to the dispatch function.

Recursive indirect calls Recursive indirect calls are a generalization of recursive direct calls in the sense that they are not restricted to functions that immediately call themselves. They share the same set of issues of recursive direct calls.

Variadic functions Variadic functions are functions that accept a variable number of parameters. They are not supported in the current implementation simply because supporting them would involve quite an amount of additional work that could not fit before the deadline.

Invoke/Unwind LLVM in addition to the *call*/*return* instructions also supports the *invoke*/*unwind* instructions that can be used to implement non-local control flow. They are not supported in the current implementation again for time constraints. They could be easily added because it would mostly involve just keeping track of a second return address for the catch pad.

Stack manipulation Functions that directly manipulates the stack and program counters (such as *setjmp*/*longjmp*) are incompatible with CGF in their standard form. Most of them can obviously be rewritten to accommodate CGF (e.g. *setjmp*/*longjmp* could be transformed in *invoke*/*unwind*). The current CGF implementation simply assumes that no function being transformed directly operates on the instruction and stack pointers.

3.3.2 Expected output

In its current implementation, the transformation takes all functions complying with the assumptions outlined in section 3.3.1 and merges them together in the *dispatch* function. It then turns the original functions in wrappers for the *dispatch* function.

The dispatch function accepts three parameters: the first is the address of the instruction to jump to when entering the function; the second is a pointer to the arguments to the function and the third is a pointer to the location where to store the return value, if present. It must be noted that the second and third arguments are pointers to locations in the stack frame of the wrapper: from this point of view, the *dispatch* function resembles a variadic function.

For each merged function, three auxiliary instructions block are created.

The first one is the outer dispatcher block: this block is the one that will start executing when the function is called from one of the wrappers and contains the code that explicitly loads the arguments from the second argument of the *dispatch* function. After the block has executed the execution continues in the inner dispatcher.

The second block is the inner dispatcher: this is the proper entry block to the function because it is the block that will start executing when the function is called from any other flattened function. As such any callsite to this function is a predecessor of this block, in fact this block is made up only of ϕ nodes gathering all the arguments and live values from all the relevant callsites (including the outer dispatcher). A special ϕ node called ϕ_{ret} is also created to contain the address of the block to execute at the end of the function. At the end of the inner dispatcher the execution jumps to the first block of the cloned function.

The third auxiliary block is the return dispatcher. This block is executed only when the corresponding outer dispatcher has been executed (i.e. the function was called through the wrapper). It stores the return value in the third argument of the *dispatch* function and terminates by returning to the wrapper.

Besides the three auxiliary blocks, each block in the original function is cloned and appended in the dispatch function. The return instruction of the function is replaced by an indirect branch to the address contained in ϕ_{ret} and call instructions are replaced by

branches to the inner dispatcher block of the relative function.

In addition to the code inside the dispatch function, the original function is turned into a wrapper that calls the dispatch function, passing the expected parameters.

4 Results

The pilot implementation discussed in chapter 3 has been used to validate many of the application ideas presented in the course of this thesis. In this chapter we present typical outputs along with explanation of various shortcomings of the LLVM framework and of the implementation itself.

4.1 Examples

Please note that while all examples presented are deliberately kept simple for the sake of discussion, the same method can be theoretically applied to complex programs.

4.1.1 Iterated calls

Let's consider the following toy code.

```
1  int a(int n) {
2      // dummy function, could be arbitrarily long
3      // or contain sub-calls
4      return n+1;
5  }
6  int b(int n) {
7      int i;
8      for (i=0; i<10000; i++)
9          n = a(n);
10     return n;
11 }
```

After translation from C to LLVM-IR and the CGF transformation, we get the CFG in figure 4.1. This CFG is transformed by subsequent optimizations in the one in figure 4.2.

In this example is already possible to see that some optimizations currently available in the LLVM framework yield suboptimal results on such simple case. Let's explain first and analyze each block of the CFG:

- %3: Entry block of the dispatch function, contains the indirect jump that transfers control to the outer dispatchers of the flattened functions.
- %10: Outer dispatcher of function *b*. Its body has been merged into %11, the outer dispatcher of function *a*, because they share the same signature.

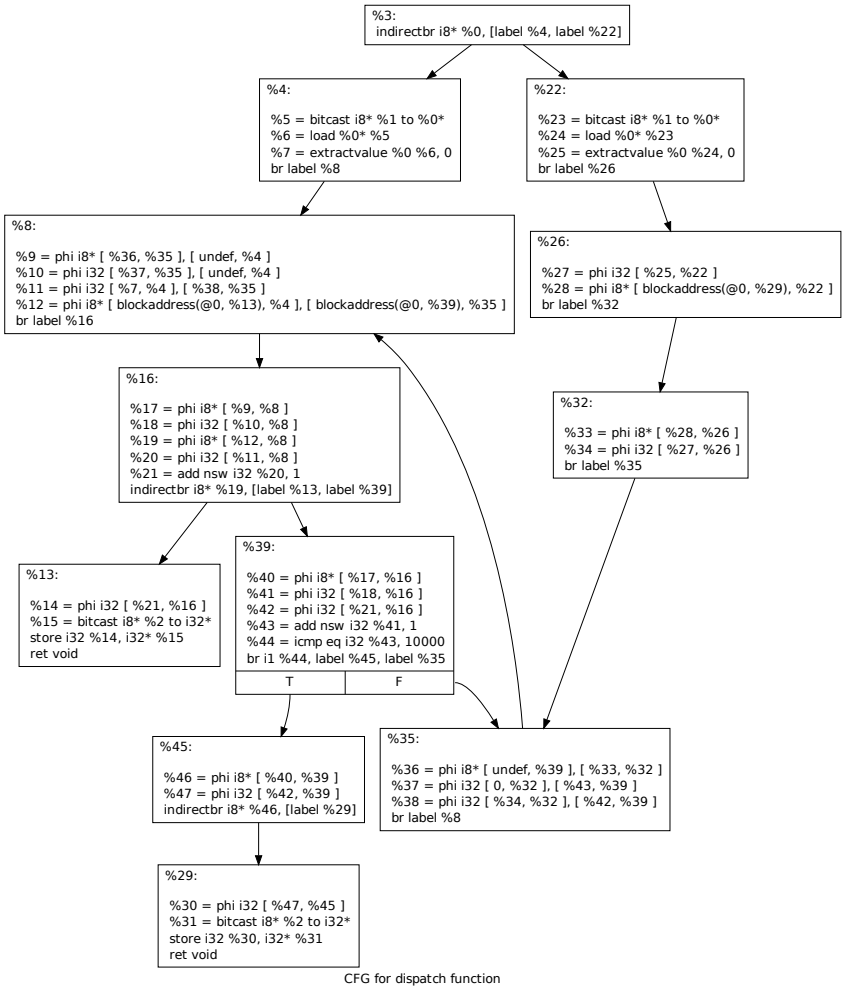

Figure 4.1: Example of CFG after transformation by the pilot CGF implementation

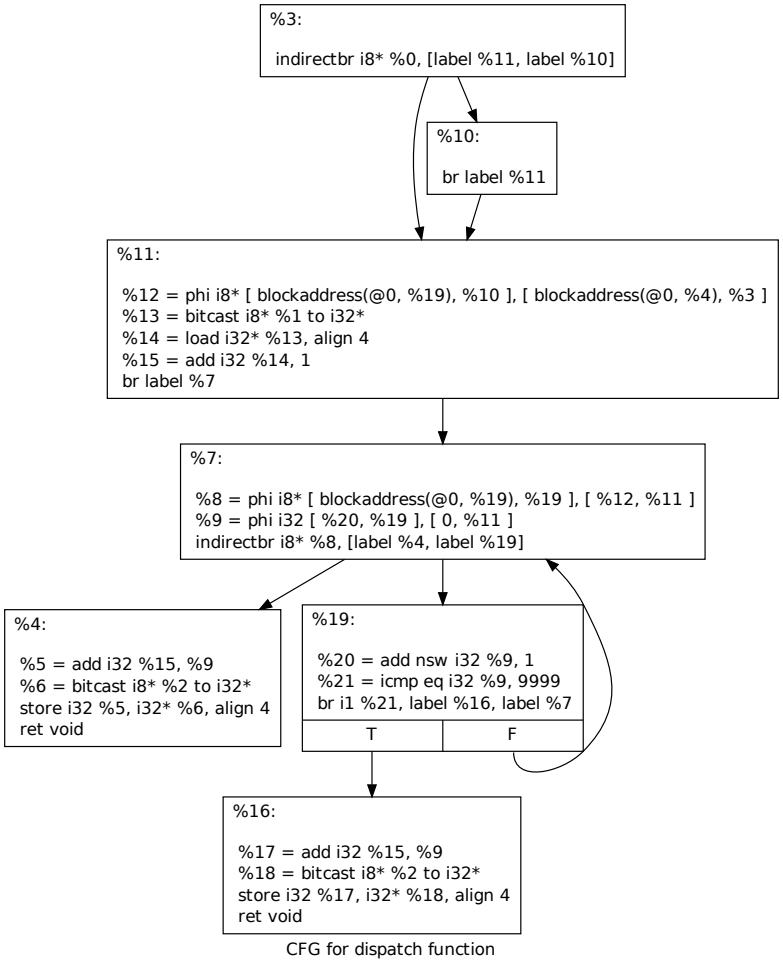

Figure 4.2: CFG of figure 4.1 after standard optimizations

- %11: Outer dispatcher of function *a*. The phi node %12 contains the address to jump when returning from a: if a was called directly control should pass to %4, the return dispatcher of function a. Otherwise it should jump to %19, the loop in function *b*. The following bitcast and load instructions read the arguments from the stack of the wrapper functions. The add instruction has been lifted from the body of function *a*.

- %7: Inner dispatcher of function *a*. The first phi, %8, tracks the return address while the second one, %9, is the loop counter of function *b*.

- %19: Loop body of function *b*.

- %4 and %16: Return dispatcher of functions *a* and *b*, respectively. The bitcast and store place on the stack of the wrapper function the value to be returned.

Missed optimizations There are a number of missed optimization opportunities that can be easily spotted:

1. Blocks %3, %10, %11 and %7 could be easily merged. This would get rid of two direct and one indirect branches.

2. The return dispatchers, %4 and %16, are identical and should be merged.

3. The loop backedge from %19 to %7 should be redirected to %19: it is easy to see that, when coming from %19 the phi node %8 the propagated value is the address of block %19 and therefore the indirectbr will always jump to %19. The redirection would also made explicit that the loop can simply be replaced by an addition by 10000. This is most likely due to the fact that current optimizations available in LLVM have problems when the terminator of the basic block being worked on is an indirect branch.

4.1.2 Multiple sequential calls

Let's take the code in the previous example and replace the loop with 4 unrolled calls to *a*. This is a rather common idiom, heavily used, for example, by overloaded C++ operators.

```
1 int b(int n) {
2     n = a(n);
3     n = a(n);
4     n = a(n);
5     n = a(n);
6     return n;
7 }
```

After compilation, we get the CFG in figure 4.3.

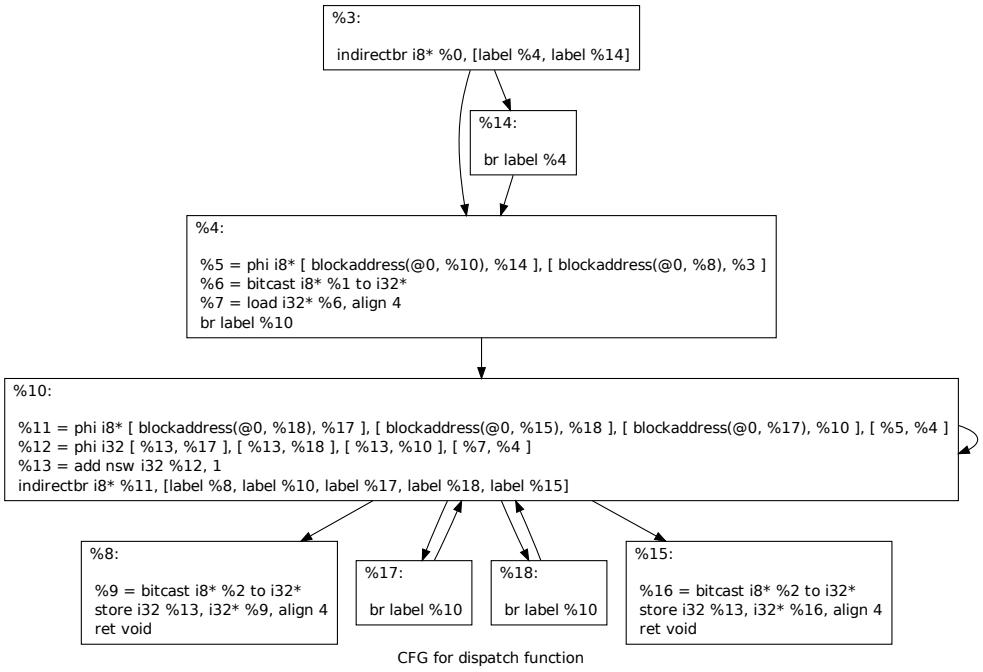

Figure 4.3: Example of CFG after CGF and standard optimizations

Again, is possible to see that the following optimizations yield suboptimal results. Similarly to previous example, blocks %3, %14, %4 and %10 could be merged together, and the same applies to blocks %8 and %15.

What's interesting in this case are blocks %10, %17 and %18. In a sense, we could say that the four calls have been "re-rolled" in a condition-less loop (more precisely, the code has been automatically threaded; see 2.2.4 for details).

To understand what this means, it is beneficial to peek at the assembler representation of the dispatch function[1]:

```
1   .type .Ldispatch,@function
2 .Ldispatch:
3   movl  $.Ltmp4, %eax   # store the return disp. of a in rax
4   jmpq  *%rdi           # jump to the requested outer disp.
5 .Ltmp2:                 # outer dispatcher of b
6   movl  $.LBB2_4, %eax  # store the address of ?
7 .Ltmp0:                 # outer dispatcher of a
8   movl  (%rsi), %ecx    # load the argument n in ecx
9   jmp .LBB2_4
10 .Ltmp8:                # block %17
11  movl  $.Ltmp6, %eax
12  jmp .LBB2_4
13 .Ltmp6:                # block %18
14  movl  $.Ltmp7, %eax
15 .LBB2_4:               # block %10
16  movq  %rax, %rsi
17  incl  %ecx            # n = n + 1
18  movl  $.Ltmp8, %eax
19  jmpq  *%rsi           # indirectbr
20 .Ltmp4:                # return dispatcher of a
21  movl  %ecx, (%rdx)    # store in pointer rdx the ret value
22  ret                   # in ecx and return to the wrapper
23 .Ltmp7:                # return dispatcher of b
24  movl  %ecx, (%rdx)
25  ret
```

Missed optimizations There are a number of missed optimization opportunities that can be easily spotted:

1. The two identical return dispatchers, %4 and %16, are still present in the assembler code (.Ltmp4 and .Ltmp7).

2. Indirect branches with few possible targets (e.g. valid values of rdi for the indirect

[1] reformatted and reordered for clarity

branch in the entry block are only .Ltmp2 and .Ltmp0) should be transformed in conditional jumps.

3. The automatic code threading mentioned above will not scale well because for each additional call a clone of block %17 (.Ltmp8) has to be created to store in eax the address of the following block (.Ltmp6 in this case, corresponding to block %18). The same effect could be attained by proper code threading, i.e. creating an address table and iterating over it as needed.

4.1.3 Layered calls

To stress test the pilot implementation, a fuzzer was created to simulate layered code architectures.

The fuzzer accepts three parameters: the number of levels L, the number of functions per layer F and the number of calls per function C. It then generates L layers each containing F functions; each function does some simple calculations and then calls, based on the results of the calculations, one out of C random functions in the layer below.

An example of these functions is the following:

```
1 int f_1_2(int a) { // layer 1, function 2
2   a += 1;
3   switch (a%3) {
4     case 0: a += f_0_2(a); break;
5     case 1: a += f_0_4(a); break;
6     case 2: a += f_0_6(a); break;
7   }
8   return a;
9 }
```

Additionally, a *main()* function is generated that iteratively calls all the other functions.

The callgraph of an example output of the fuzzer is shown in figure 4.4. The *external node* shown therein is a meta-node with edges towards functions that can be called from other modules.

To show the difference between regular compilation and compilation using CGF it is beneficial to visualize the CFG of the same module under the two different approaches. Figure 4.5 show the resulting CFGs side-by-side. A detailed analysis of missed optimization opportunities in this scenario, especially for high values of L, is given in section 4.2.

4.1.4 Forwarding wrappers

Forwarding wrappers are auxiliary functions provided for binary compatibility with other C code and to support all the constructs that are currently unsupported by the pilot implementation (see section 3.3.1). As such they have not been implemented to be fast or compact.

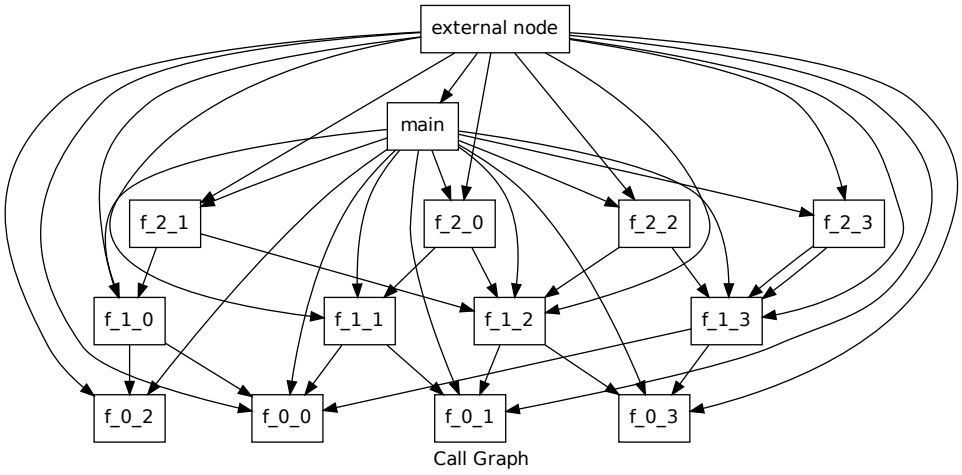

Figure 4.4: Example of callgraph generated by the fuzzer with parameters $L = 3, F = 4, C = 2$

As an example consider the following assembler representation of the wrapper of function a in the first example:

```
1   .type a,@function
2   subq  $24, %rsp        # allocate space on the stack
3   movl  %edi, 16(%rsp)   # store the argument n on the stack
4   movl  $.Ltmp0, %edi    # address of the outer dispatcher
5   leaq  16(%rsp), %rsi   # address of the incoming argument(s)
6   leaq  12(%rsp), %rdx   # address of the return value(s)
7   callq .Ldispatch       # call to the dispatch function
8   movl  12(%rsp), %eax   # load the ret value from the stack
9   addq  $24, %rsp        # deallocate space on the stack
10  ret                    # return
```

Currently, one such function is created for all flattened functions. This not strictly needed because functions with the same signature could share most of the code (the only differing element between wrappers is the address of the outer dispatcher).

To be useful, a better way of passing the arguments and return values between wrappers and dispatch should be designed. Promising alternatives include the use of the LLVM trampoline intrinsic[46] or other adaptive methods.

It should also be considered that if binary compatibility is not required and the wrapper can be inlined (this happens e.g. in case of flattened recursive functions) much of the overhead can be avoided.

For all these reasons, the code size figures in section 4.2 do not include the wrappers.

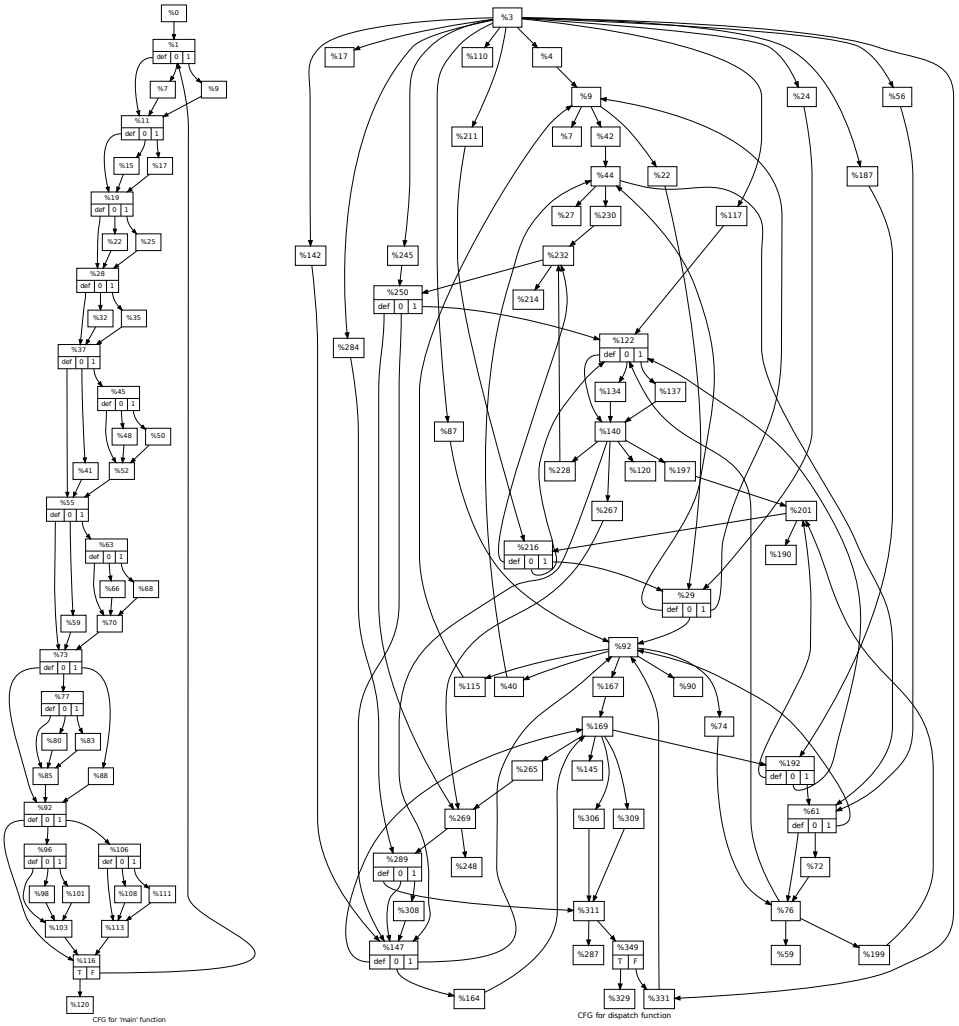

Figure 4.5: CFGs of the module in figure 4.4 with different optimization strategies Both CFGs have been optimized using -O3. The one on the right has been preprocessed with CGF. Note that the CGF-preprocessed version contains the code needed to run any of the functions in the original module, whereas in the one on the left only the main function is available (all calls have been inlined with the resulting code duplication; the clusters still visible are the original functions, four on top and four at the bottom).

4.2 Benchmarks

As highlighted in the previous section, while many of the optimizations currently available in LLVM already do an egregious work on CGF-transformed code, there are many cases where the produced code is suboptimal. This is likely due to both weaknesses in the LLVM optimization framework and to lack of tuning and verbosity of CGF-transformed code.

Moreover, since most real-world code use features not currently supported by our pilot implementation, performance-related figures would be pretty much meaningless.

For these reasons we focused our attention to measuring code size. As discussed in section 2.1, literature results of similar attempts resulted in an average code size increase of 13% when targeting 8-bit microcontrollers.

Using our fuzzer we proceeded to generate different random programs with varying parameters. The programs were compiled using the same compiler and optimizations, to measure the effects of CGF on code size. All tests were done on a box running Ubuntu 11.04 x64.

For each compiled program we measured the size of the object size (considering only the sections used by instructions). Additionally, when compiling using CGF also the number of register spills and the allocated stack size were recorded.

Each program was compiled three times with different switches: -O3, -no-internalize -O3 and -cgf -O3. The -no-internalize switch instructs the compiler to not discard functions that are not used by *main()*: the result is that all functions originally present in the module are still available after the module has been compiled. Without this switch the only function present after compilation is *main()*. For code compiled with the -cgf switch we measured both the size of the compiled code with and without the wrappers. As explained in section 4.1.4, the wrappers are currently not optimized and tend to take a lot of space. It is important to note that the flattened functions are still present and available to external callers[2] even without the forwarding wrappers[3].

Table 4.1 lists the average resulting code sizes for code generated using different parameter of the fuzzer. All values are expressed in bytes. The "Δ" column is the difference between the "-O3" column and the "-cgf -O3 (no wrappers)" columns. The "-O3 (main only)" can be roughly considered a lower bound on the possible sizes of code compiled with -O3, either with or without CGF.

From this table it is possible to see that different fuzzer parameters yield completely different size differences.

We can identify three groups of parameters. The first group is made up of modules with few layers and many functions. This modules tend to grow a little (~10%) or even shrink as a consequence of CGF. This is mostly due to the fact that the size cost of missed optimizations caused by CGF is smaller than the size cost caused by having many small regular functions.

[2] indeed it could be argued that CGF-transformed code is simply akin to code having a different calling convention

[3] the fuzzer does not generate recursive or indirect calls, therefore the program executes correctly without wrappers

Fuzzer parameters	-O3 (main only)	-O3	-cgf -O3 (no wrappers)	-cgf -O3	Δ
2-64-4	3433	8207	7420	13564	-9.5%
4-4-2	1632	3136	3354	4122	6.9%
2-16-4	948	2129	2381	3917	11.9%
2-4-2	174	410	559	943	36.3%
2-2-2	106	230	338	530	47.3%
8-4-2	4193	6522	13611	15147	108.7%
4-8-4	5328	6663	53623	55159	704.7%

Table 4.1: Average compiled code size for different fuzzer parameters

Fuzzer parameters	Stack usage	Spills	Functions	Callsites	Δ
2-64-4	24	8	129	384	-9.5%
4-4-2	121.6	42	17	40	6.9%
2-16-4	24	10	33	96	11.9%
2-4-2	24	0	9	16	36.3%
2-2-2	24	0	5	8	47.3%
8-4-2	542	137	33	88	108.7%
4-8-4	1350	319	33	128	704.7%

Table 4.2: Average properties of CGF-transformed code for different fuzzer parameters

The second group is made up by modules having very few functions. In this case the code size increases moderately (~40%): this is due to the fact that the size cost of CGF is higher than that of having separate functions. In particular, the number of functions and the quantity of code is so small to be comparable to the amount of auxiliary code of the dispatch function. In section 4.1 we have already seen that better optimizations and a better implementation of CGF should be able to drastically lower the amount of auxiliary code.

The last group is made up by functions with many layers and many functions. In this case the increase of code size is huge (>100%). To understand why this happens it's necessary to turn the attention to the CPU architecture.

Table 4.2 shows some key properties of the compiled code. Stack usage is the size in bytes of the frame allocated on the stack for the dispatch function and represents an upper bound on the number of bytes used by the function[4]. Spills are the number of registers spilled due to excessive register pressure. Functions and callsites are, respectively, the number of functions and callsites in the module.

From this second table is possible to understand why the third group discussed above

[4]the lower bound of 24 bytes is due to the fact that the dispatch function accepts as arguments 3 pointers (the benchmarks have been run on a 64bit machine); of these 24 bytes, 16 can be used by the functions because their value dies as soon as the outer dispatcher finishes executing

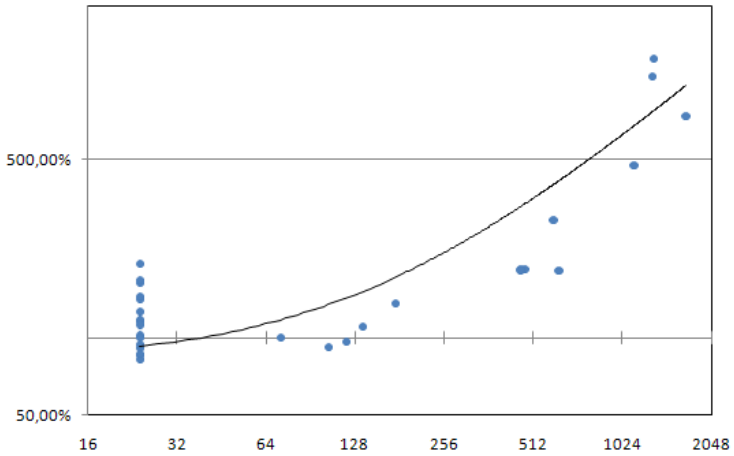

Figure 4.6: Impact of register spilling on code size
Relative ode size (Y axis) as a function of the frame size (X axis). Both axis are logarithmic. The regression line is linear.

behaves so badly. The reason of this resides in the fact that these function need lots of space to spill the registers during execution. While this per se can't explain the code size explosion, it can hint to the real reason.

By examining the resulting assembler code it is in fact possible to see that in these pathological cases lots of spills and reloads are not done to load a value in a register, but rather to shuffle the spilled values around on the stack. This is caused by the fact that the current register spiller available in LLVM is designed to operate in functions, and therefore takes in consideration mostly the needs of the current block and immediate neighbors. The result of this deficiency is that the increase in code size is linear with the size of the stack frame used by the dispatch function (figure 4.6). Moreover, poor choices in register spilling also cause a further increase in the size of the stack frame. These problems were discussed in section 2.4.6.

To produce efficient CGF code the spiller[5] should instead operate globally, taking into account the context-sensitive CFG, the relative path usage and the results of data-flow analysis. There are discussions underway in the LLVM community to improve or replace the current spiller and register allocator to allow better performance in complex cases such as the code generated by CGF. This is absolutely needed to be able to use CGF on large code bases.

In addition to the above problems, the current pilot implementation tends to produce overly verbose and redundant ϕ nodes. When coupled with the fact, already mentioned in the examples section, that current optimizations available in LLVM are impaired when dealing with indirect branches, this tends to further exacerbate the above problems.

As a side note, it is worth mentioning that the pathological results obtained on our

[5]this is true for the register allocator as well; in our testing, though, the primary cause of inefficient code could be traced back to the spiller

architecture (x64) could very well be very different on other architectures. In fact, x86 and x64 are architectures with small amounts of registers: other architectures, such as ARM and Itanium, are less susceptible to such problems due to the availability of bigger register files.

If we focus on the first group (the only one whose benchmarks have currently some significance due to the highlighted problems in LLVM) we see that the results of our pilot implementation are slightly better, as far as code size is concerned, than those found in literature. This is very promising, because there are wide margin of improvements in the form of better (or simply slightly tuned) optimizations and better design decisions for the CGF implementation.

5 Conclusions

Automatic compiler optimizations is a complex topic. This is especially true when writing architecture-agnostic optimizations because we can't take into account the peculiarities of the final platform and also because the runtime performance of the compiled code is a product of the complex interactions of many different optimizations.

Nevertheless, given the rising importance and spread of information processing systems, especially mobile ones, the need for effective tools that empower improvements of the capabilities of such systems is more pressing than ever. As such, further research in the field of compiler optimizations is a priority.

In this thesis we analyzed the applicability and pilot implementation of call-graph flattening (CGF), a code transformation that gets rid of the abstraction (and overhead) provided by traditional functions in compiled code. As such, CGF does not constitute an optimizing transformation but rather an optimization-empowering transformation, i.e. a transformation that enables other optimizations to perform better. Despite this, literature results indicate that substantial reductions in stack usage and increase in performance can be obtained.

In particular, moving from previous studies, this thesis first analyzed in depth the relations between CGF and related concepts such as CPS. We then proceeded to study which scenarios would benefit more from the optimizations empowered by CGF and which ones would prove problematic. For the latter we proposed known solutions from the literature or suggested further research directions. Finally, we described a pilot implementation of CGF based on the LLVM framework. Even if this implementation is still limited to simple cases and is not able to cope with real-world use cases, it shows great potential even if the other optimizations presently available in the LLVM framework are unable to work effectively on the unusual constructs produced by CGF. Our initial benchmarks indicate that the current pilot implementation can already perform slightly better than the results in literature.

For all the above reasons, the author plans to keep working on the subject (and the further research directions outlined in the course of the thesis) to contribute upstream a complete working implementation and to publish more comprehensive and definitive results.

Finally, the author would like to thank his supervisor - professor Silvano Rivoira - for his suggestions, support and for granting the freedom to tackle this important topic. The author would like to thank as well Marco Sgrignuoli for his reviews of the preliminary drafts of this thesis.

Bibliography

[1] Wikipedia. Ubiquitous computing — wikipedia, the free encyclopedia, 2011. [Online; accessed 5-June-2011]. 1.2

[2] Wikipedia. Compiler optimization — wikipedia, the free encyclopedia, 2011. [Online; accessed 3-May-2011]. 1.3, 1.3.1

[3] Wikipedia. Inline expansion — wikipedia, the free encyclopedia, 2010. [Online; accessed 24-May-2011]. 1.3.1.1, 2.3.2

[4] Wikipedia. Interprocedural optimization — wikipedia, the free encyclopedia, 2011. [Online; accessed 30-June-2011]. 1.3.1.1

[5] Wikipedia. Automatic parallelization — wikipedia, the free encyclopedia, 2011. [Online; accessed 30-June-2011]. 1.3.1.2

[6] Guiseppe Messina Geoffrey Fox, Roy Williams. *Parallel Computing Works!*, pages 575–593. Morgan Kaufmann, 1994. 1.3.1.2

[7] Jaewook Shin. Introducing control flow into vectorized code. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:280–291, 2007. 1.3.1.2, 2.4.5.2

[8] Wikipedia. Static program analysis — wikipedia, the free encyclopedia, 2011. [Online; accessed 19-May-2011]. 1.3.2

[9] Brian J. Gough. *An Introduction to GCC - for the GNU compilers gcc and g++*, chapter 6.1.2. 2005. 2

[10] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler, 2011. 2, 2.4.7.1

[11] K.D. Cooper, M.W. Hall, and L. Torczon. An experiment with inline substitution. *Software: Practice and Experience*, 21(6):581–601, 1991. 2, 2.3.2

[12] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, pages 146–160, 1989. 2, 2.3.2

[13] Wikipedia. Trampoline (computing) — wikipedia, the free encyclopedia, 2011. [Online; accessed 19-May-2011]. 4

[14] Wikipedia. Shim (computing) — wikipedia, the free encyclopedia, 2011. [Online; accessed 19-May-2011]. 4

[15] G.L. Steele Jr. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 annual conference*, pages 153–162. ACM, 1977. 2.1, 2.2.2, 2.4.6.1

[16] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997. 2.1, 2.5.1, 28

[17] A. Majumdar, C. Thomborson, and S. Drape. A survey of control-flow obfuscations. *Information Systems Security*, pages 353–356, 2006. 2.1, 2.5.1

[18] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *acsac*, page 308. Published by the IEEE Computer Society, 2000. 2.1, 2.5.1

[19] A. Venkatraj. *Program obfuscation*. PhD thesis, MS Thesis. Department of Computer Science, University of Arizona, 2003. 2.1, 2.5.1

[20] Xuejun Yang, Nathan Cooprider, and John Regehr. Eliminating the call stack to save ram. *SIGPLAN Not.*, 44:60–69, June 2009. 2.1, 2.4, 2.5.2, 2.5.3, 2.5.4, 3.3

[21] G.J. Sussman and G.L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. 2.2.1

[22] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 13–22, New York, NY, USA, 1995. ACM. 2.2.1

[23] Wikipedia. Continuation-passing style — wikipedia, the free encyclopedia, 2011. [Online; accessed 4-May-2011]. 2.2.1

[24] A.W. Appel. *Compiling with continuations*. Cambridge Univ Pr, 2007. 2.2.1

[25] Wikipedia. Tail call — wikipedia, the free encyclopedia, 2011. [Online; accessed 30-June-2011]. 2.2.2

[26] POSIX. Man page for setjmp(3), 2011. [Online; accessed 4-May-2011]. 2.2.3

[27] Wikipedia. Setjmp.h — wikipedia, the free encyclopedia, 2011. [Online; accessed 4-May-2011]. 2.2.3

[28] Wikipedia. Threaded code — wikipedia, the free encyclopedia, 2011. [Online; accessed 4-May-2011]. 2.2.4

[29] Wikipedia. Call stack — wikipedia, the free encyclopedia, 2011. [Online; accessed 24-May-2011]. 2.3.3.2

[30] Jason Evans. Freebsd - malloc.c, 2011. 2.3.3.2

[31] G. Aigner and U. Holzle. Eliminating virtual function calls in c++ programs. *ECOOP '96 - Object-Oriented Programming*, page 142, 1996. 2.3.3.3

[32] B. Calder and D. Grunwald. Reducing indirect function call overhead in c++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408. ACM, 1994. 2.3.3.3

[33] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *INFORMATICS: 10 YEARS BACK, 10 YEARS AHEAD*, pages 86–101. Springer, 2000. 2.3.3.6

[34] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003. 2.3.3.6

[35] Wikipedia. Language-based system — wikipedia, the free encyclopedia, 2011. [Online; accessed 8-May-2011]. 2.3.3.6

[36] Wikipedia. Common subexpression elimination — wikipedia, the free encyclopedia, 2011. [Online; accessed 24-May-2011]. 2.3.5

[37] Clinton F. Goss. Machine code optimization - improving executable object code, 1986. 2.3.5

[38] Clinton F. Goss Martin Charles Golumbic, Robert B. K. Dewar. Macro substitutions in micro spitbol - a combinatorial analysis. In *Proc. 11th Southeastern Conference on Combinatorics, Graph Theory and Computing, Congressus Numerantium, Utilitas Math.*, pages 485–495, Winnipeg, Canada, 1980. 2.3.5.1

[39] Wikipedia. Vectorization (parallel computing) — wikipedia, the free encyclopedia, 2011. [Online; accessed 30-June-2011]. 2.4.5.2

[40] Brian J. Gough. *An Introduction to GCC - for the GNU compilers gcc and g++*. 2005. 2.4.6.4

[41] Agner Fog. Optimizing subroutines in assembly language. pages 85–86. 2011. 2.4.7.2

[42] Agner Fog. Optimizing subroutines in assembly language. page 87. 2011. 2.4.7.2

[43] Wikipedia. Return-to-libc attack — wikipedia, the free encyclopedia, 2010. [Online; accessed 19-May-2011]. 30

[44] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. 3.2

[45] V. Makarov. Llvm 2.7 vs gcc 4.5.0 performance benchamark (x86), 2010. 3.2

[46] Chris Lattner. Trampoline intrinsic - llvm language reference manual, 2011. 4.1.4

Fuzzer source code

```
var maxidx = 4;
var maxlevel = 3;
var maxspread = 2;
function create(level, idx, spread) {
    var str = "int␣f_"+level+"_"+idx+"(int␣a)␣{\n";
    str += "␣␣a␣+=␣" + Math.floor(Math.random()*5-2) + ";\n";
    if (level > 0 && spread > 0) {
        str += "␣␣switch␣(a%"+spread+")␣{\n";
        for (var i=0; i<spread; i++) {
            var ridx = Math.floor(Math.random()*maxidx);
            str += "␣␣␣␣case␣"+i+":␣a␣+=␣f_" + (level-1) +
                   "_" + ridx +"(a);␣break;\n";
        }
        str += "␣␣}\n";
    }
    str += "␣␣return␣a;\n}\n";
    return str;
}

var str = "";
var str2 = "";
for (var i=0; i<maxlevel; i++) {
    for (var j=0; j<maxidx; j++) {
        str += create(i, j, maxspread);
        str2 += "␣␣␣␣j␣=␣f_"+i+"_"+j+"(j);\n"
    }
}
str += "int␣main()␣{\n";
str += "␣␣int␣i,␣j␣=␣0;\n";
str += "␣␣for␣(i=0;␣i<1<<20;␣i++)␣{\n";
str += "␣␣␣␣j␣+=␣i;\n";
str += str2;
str += "␣␣}\n";
str += "␣␣return␣j;\n";
str += "}\n";
print str;
```

CGF-transformed CFG generated by Fuzzer-8-8-4