Architecture for a Next-Generation GCC

Chris Lattner Vikram Adve
University of Illinois at Urbana, Champaign
{lattner, vadve}@cs.uiuc.edu
http://llvm.cs.uiuc.edu

Abstract

This paper presents a design and implementation of a whole-program interprocedural optimizer built in the GCC framework. Through the introduction of a new language-independent intermediate representation, we extend the current GCC architecture to include a powerful mid-level optimizer and add link-time interprocedural analysis and optimization capabilities. This intermediate representation is an SSA-based, low-level, strongly-typed, representation which is designed to support both efficient global optimizations and high-level analyses. Because most of the program is available at link-time, aggressive "whole-program" optimizations and analyses are possible, improving the time and space requirements of compiled programs. The final proposed organization of GCC retains the important features which make it successful today, requires almost no modification to either the front- or back-ends of GCC, and is completely compatible with user makefiles.

1 Introduction

The GNU Compiler Collection (GCC) [15] is in many ways the centerpiece of the Free Software movement. It supports several source languages and a plethora of back-ends for various targets, providing a unified target for free software. GCC has been successful because of its extreme portability, stability, and because it is able to compile and optimize several popular source languages (C, C^{++}, Java, etc) to each target. Unfortunately, despite the success of the GCC compiler suite as a whole, the optimization infrastructure is still not competitive with commercial compilers.

Over the years, the GCC optimizer has evolved from compiling a statement at a time, to compiling and optimizing entire functions at a time, to the (still very new) support for unit-at-a-time compilation (compiling and optimizing all of the functions in a translation unit together). As the scope for analysis and optimizations increases, the compiler is better able to reduce the time and space requirements for the generated code.

This paper proposes the next logical step for the GCC optimizer: extend it to be able to analyze and optimize *whole programs* at link-time[1], enabling new optimizations and making existing analyses and optimizations more powerful. For example:

- inlining across translation units
- whole-program alias analysis
- interprocedural register allocation
- interprocedural constant propagation
- data layout optimizations
- exception handling space optimizations
- sorting initializer priorities at link-time

The key challenges to whole-program optimization are to enable powerful transformations while keeping compile times reasonable, and to keep the user-visible development process unchanged (e.g. user makefiles).

The architecture that we propose is based on a new language-independent low-level code representation that preserves important type information from the source code. The use of a low-level, SSA-based representation allows the compiler to perform a variety of optimizations at compile time, off-loading work from the link-time optimizer. However, the link-time optimizer can only perform meaningful optimizations on the program if it has enough high-level information about the program to prove that aggressive optimizations are safe. Because of this, the low-level code representation is typed (using a language-independent constructive type system) and directly

[1]This capability would be optional and could be enabled only when the program is compiled at the "-O4" level of optimization, for example.

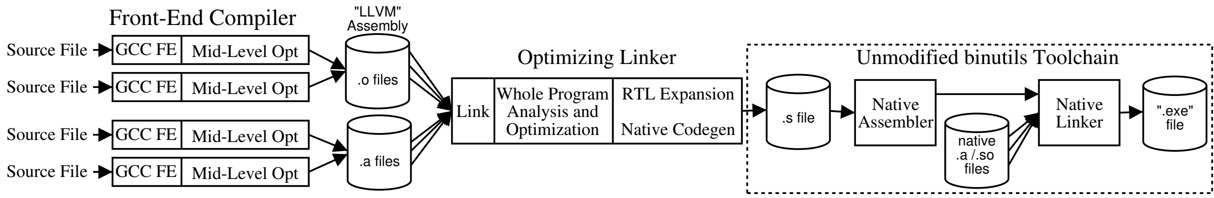

Figure 1: High-Level Compiler Architecture for Whole-Program Optimization

exposes information about structure and array accesses to the optimizer.

The link-time optimizer is designed to combine the translation units of a program together and do the final whole-program optimization. After the program is optimized, machine code is generated at link-time for the entire program at once, allowing a variety of interprocedural low-level code optimizations to be performed.

The Low-Level Virtual Machine (LLVM) [10] is an implementation of the architecture and intermediate representation [11] described in this paper, which allows us to be more concrete when describing aspects of the design. This system has served as the host for several research projects [7, 13, 12] which require whole-program information as well as a host for a variety of traditional compiler optimizations.

We hope that the lessons learned by the LLVM project will be useful to the GCC community, and are willing to contribute as much code to the GNU project as there is interest in. We are planning to have our first public release of LLVM, with a liberal license, in the Summer of 2003. However, LLVM will only be discussed when it helps clarify the ideas in the proposed architecture, this paper is intended to be a GCC paper, not an LLVM paper.

This paper is organized as follows: Section 2 describes the proposed high-level architecture in detail, including modifications that would need to be made to the GCC infrastructure. Section 3 describes important aspects of the proposed intermediate representation for the system. Section 4 describes LLVM, our existing implementation of the proposed design. Section 5 describes other work related to the proposed design, and Section 6 wraps up the paper.

2 High-Level Compiler Architecture

The proposed high-level architecture is illustrated in Figure 1. The essential aspect of this design is that it separates the current `cc1` program into two components: a front-end compiler and an optimizing linker. The front-end retains all of the responsibilities of current GCC front-ends (preprocessing, lexical analysis, parsing, semantic analysis, etc..) and should work unmodified in the new system. After each function is parsed and checked for semantic errors it is "expanded" from the "tree" representation to the new language-independent intermediate representation (described in Section 3). Once the entire translation unit has been translated (and if no errors have occurred), a standard set of mid-level optimizations are performed on the translated module. After these optimizations are finished, a "`.o`" file is emitted which contains IR assembly code for the representation.

When the optimizing linker is invoked, it reads in all of the translated IR files and any libraries compiled to the intermediate representation. It links these files together into a single-file representation of the program, on which it can run whole-program analyses and optimizations. Finally, once these analyses and transformations are complete, the GCC backend is invoked to expand the intermediate representation into RTL and use the configured target description to produce a native `.s` file.

After the optimizing linker produces a native `.s` file, the compilation process proceeds through the standard system assembler and linker (to resolve any symbols in libraries that were not available in the IR form), finally producing a native executable.

2.1 Compatibility and Implementation

One of the key features of this design is that it is compatible with the standard "compile and link" models of compilation, and is thus fully compatible with existing makefiles. In order to provide this compatibility, the link phase of the `gcc` compiler driver is extended to invoke the optimizing linker and system assembler (if necessary) during the standard link step of the compile process. In this way, any input files that are in the IR format are automatically linked together and optimized without interfering with the compilation and linking of standard translation units and libraries. If no files in the IR format are present, the entire invocation of the optimizing linker is skipped.

Another important aspect of the design is how the compiler works when whole-program optimization is not enabled. If not enabled, each translation unit is either compiled a function at a time or a unit at a time (depending on the setting of the `-funit-at-a-time` switch), through the mid-level optimizer, RTL expansion, and code generation phases of the compiler. This produces a native .s file, which can be processed with the standard system assembler and linker, as before.

For this approach to be feasible, a large amount of code must be shared between the optimizing linker and the compiler front-ends. This can either be accomplished through the use of libraries that are shared between the two (which would contain the existing GCC back-end, and any shared optimizations on the IR), or by making both logical pieces be part of the same binary. In either case, the actual organization of the existing GCC code base would not have to change in any substantial way.

2.2 Architectural Issues Affecting Performance

In addition to providing the desired functionality and compatibility with existing systems, it is crucial that the compiler does not slow down unacceptably — even if whole-program optimization is only enabled at `-O4`. In practical terms, this design addresses the issue by performing as much optimization as possible at compile time.

Any time a source file is changed, it must be recompiled and the application must be relinked. In order to reduce the amount of work that must be done, this design allows most traditional optimizations to be performed in the compiler front-end stage, rather than requiring all optimization to occur at the link stage (as is common for whole-program optimizers). Because most aggressive scalar optimizations are performed at compile-time, they would not need to be rerun at link time, reducing the time for compilation. Of course, the compiler performance issue does not even arise unless the user is modifying the program and recompiling at `-O4`.

Optionally with this design, the compiler could try to minimize the amount of recompilation necessary when a change occurs by keeping track of which interprocedural information is used to modify functions in other translation units, building a dependence graph between the modules [4]. In practice, however, this would make the compiler much more complicated and prone to subtle bugs that are hard to reproduce. We feel that although the cost of recompilation is still fairly substantial in our system (native code must be regenerated for the entire application), that the extra complexity introduced into the compiler must be weighed against the recompilation time penalties, and thus may be impractical.

3 Code Representation

The representation used to analyze and manipulate the program determines what kinds of transformations are possible and when in the compilation process they must be performed to be successful. As mentioned earlier, we propose using a language-independent, low-level, SSA-based, strongly-typed representation as the sole representation used for the mid-level and link-time optimizers. This representation is a first-class assembly language, which includes all of the information necessary to represent the program (and is in fact directly interpretable). Concrete details of the representation used by LLVM are included in Section 3.2.

Using a low-level three-address code representation based on Static Single Assignment [6] form enables the direct application of many well-known and efficient global optimizations. SSA form permits *sparse* optimizations that do not, in general, require bitvector data-flow analysis to compute results. Using a three-address code representation (as opposed to an tree structured representation) also makes transformations easy to develop and reason about.

Many transformations need information about the high-level behavior of the program to be effective. In order to preserve this information, we propose that the representation maintain a strong (but language-neutral) type system, which captures information about pointer, structure, and array accesses in the program. Working with the LLVM system we find that this type information allows for a variety of high-level analyses and transformations [7, 13, 12] while the nature of the low-level representation makes it very easy to manipulate. Another advantage of type information is that it makes detecting and understanding bugs in transformations much easier.

The goal of the program representation is to enable as many different types of optimizations as possible. Because of this, it is important that the representation be able to represent *all parts* of a program (including global variables, and file scope `asm` statements, for example) in a form that allows transformations to modify it. Another useful feature of the representation is a stable textual format ("assembly language") that can be read and written by the compiler. Given this, it is trivial to write unit tests for transformations and to debug transformations in isolation from the rest of the compiler, and the representation can be directly interpreted for immediate feedback on a transformation.

3.1 Performance Aspects of the Representation

Once the optimizing linker brings together the compiled program into one module, the interprocedural analysis and optimization passes are used to improve the program. Because these passes operate on the entire program at once, however, the efficiency of each analysis or optimization is critical. For this reason, several aspects of the representation are designed to make these transformations as efficient as possible.

In particular, the use of an SSA-based representation allows for efficient, sparse, global optimizations, and can make flow-sensitivity much less important in many analyses (reducing cost substantially). In addition, the three-address code representation has a small memory footprint and simple memory ownership semantics (eliminating the need for it to live on a garbage collected heap). In our experience with LLVM, code optimizers for a sparse representation can be several times faster than optimizations on a dense representation like RTL.

3.2 A Concrete LLVM Example

Figure 2 gives an example of a C function and the corresponding LLVM module it compiles to. The example shows several important aspects of the LLVM representation. In particular, it gives a simple example of the type system, basic instruction flavor, and demonstrates some instructions. More details about the LLVM representation can be found in the LLVM language reference [11].

LLVM uses a simple constructive type system composed of primitive types, structures, arrays, and pointers. Although this is a very simple type system, we believe that it contains the key features necessary for a front-end to lower any high-level type onto it. For example, the LLVM C^{++} front-end lowers classes with inheritance into nested structure types. Types are very important in the LLVM system, and everything that can be used as an operand to an instruction has a type.

Functions in LLVM contain a list of basic blocks, and each basic block contains a list of instructions. LLVM has only 29 instructions, which include standard instructions like `load`, `xor`, `set`*cc*, etc and a `phi` instruction for representing SSA form[2]. Intraprocedural control flow in LLVM is very simple (consisting of conditional branches, unconditional branches, and the `switch` instruction).

Everything in LLVM is explicit: there are no fall-through branches, all address arithmetic is exposed (at the level of structures, pointers, and arrays), and all references to memory use the `load` and `store` instructions. This makes the language more uniform and simple to analyze and transform.

The `getelementptr` instruction in LLVM provides the mechanism for structured address arithmetic[3]. The `getelementptr` instruction is exactly analogous to sequences of array subscript and structure

[2] SSA ϕ-nodes are eliminated during the register allocation phase of native code generation.

[3] LLVM code can also cast a pointer to an integer type, add an arbitrary offset to it, then cast it back to a pointer, if unstructured address arithmetic is necessary.

```c
typedef struct QuadTree {
  double Data;
  struct QuadTree *Children[4];
} QT;

void Sum3rdChildren(QT *T,
                    double *Result) {
  double Ret;
  if (T == 0) { Ret = 0;
  } else {
    QT *Child3 =
      T[0].Children[3];
    double V;
    Sum3rdChildren(Child3, &V);
    Ret = V + T[0].Data;
  }
  *Result = Ret;
}
```

(a) Example function

```llvm
%struct.QuadTree = type { double, [4 x %QT*] }
%QT = type %struct.QuadTree

void %Sum3rdChildren(%QT* %T, double* %Result) {
entry:  %V = alloca double              ;; %V is type 'double*'
        %tmp.0 = seteq %QT* %T, null    ;; type 'bool'
        br bool %tmp.0, label %endif, label %else

else:   ;;tmp.1 = &T[0].Children[3]   'Children' = Field #1
        %tmp.1 = getelementptr %QT* %T, long 0, ubyte 1, long 3
        %Child3 = load %QT** %tmp.1
        call void %Sum3rdChildren(%QT* %Child3, double* %V)
        %tmp.2 = load double* %V
        %tmp.3 = getelementptr %QT* %T, long 0, ubyte 0
        %tmp.4 = load double* %tmp.3
        %tmp.5 = add double %tmp.2, %tmp.4
        br label %endif

endif:  %Ret = phi double [ %tmp.5, %else ], [ 0.0, %entry ]
        store double %Ret, double* %Result
        ret void   ;; Return with no value
}
```

(b) Corresponding LLVM code

Figure 2: C and LLVM code for a function

index expressions, returning the address of the last element indexed[4]. For example, the `%tmp.1` instruction in Figure 2(b) first indexes into the 0^{th} element from the pointer, then into the 1^{st} structure element (the "Children" member), then into the 3^{rd} element of the array. Structured address arithmetic exposes the necessary high-level information about structure and array accesses directly to analyses and transformations which need it.

One important aspect of the LLVM language is that all references to memory happen with `load` and `store` instructions, and that there is no "address-of" operation. In LLVM, all objects which live in memory (global variables, functions, the heap, and the stack) are explicitly allocated and exposed by their address, not their value. In Figure 2, for example, the `V` variable is required to live in memory so that its address may be passed into a recursive invocation of `Sum3rdChildren`. Because it is impossible to take the address of a virtual register, stack memory must be explicitly allocated with the `alloca` instruction[5], and any references to `V` must use `load` and `store` instructions. This dramatically simplified def-use chain construction for virtual registers, which would otherwise require some form of alias-analysis to construct.

A final example illustrating how LLVM simplifies the development of transformations is the operators that it lacks. In particular, LLVM does not have (or need) any unary operators or a copy instruction. Instead of providing the standard negate and bitwise complement unary operators, LLVM represents these with standard binary operators where one operand is a constant ("`neg x`" = "`sub 0, x`" and "`not x`" = "`xor x, -1`"). This reduces the dependence on a "canonical form" for the representation and simply reduces the number of instructions that need to be handled.

The lack of a copy instruction is possible through the use of SSA form, and because def-use chains are trivially computed and always available. Any time a copy instruction would be inserted (to replace a redundant computation for example) it is sufficient to replace any uses of the destination with uses of the source operand (by following the def-use chains), implicitly performing copy propagation automatically. This simple feature has actually avoided several phase-ordering issues that would otherwise require unnecessary passes over the representation to do copy propagation between other passes.

4 LLVM Compiler Infrastructure

The LLVM Compiler Infrastructure [10] currently consists of approximately 130,000 lines of C++ code and a the front-end, which is a patch against the mainline GCC CVS tree. This code largely implements the design presented in this paper, al-

[4] The example in Figure 2(a) uses the strange syntax '`T[0].x`' instead of using the equivalent '`T->x`' to make the correspondence more clear.

[5] When the back-end is invoked, all fixed sized `alloca`s in the entry block are treated the same as address-exposed automatic variables.

though there are some differences. This section describes these differences, the implementation status of LLVM, some other features of LLVM that make writing transformations simpler, and some insights that we have had while working on LLVM.

4.1 Implementation Status

The LLVM C front-end is based on the mainline GCC CVS repository. It generates code by calling LLVM versions of functions that are equivalent to the RTL-expansion routines (e.g. `llvm_expand_expr`, `llvm_expand_function_start`, `make_decl_llvm`, etc) during compilation. These routines build up an LLVM version of the translation unit, which is then written to the "`.s`" file all at once (allowing "unit-at-a-time" style transformations to be performed from within GCC in the future).

Instead of modifying the `cc1` binary to interface directly to the LLVM optimizations written in C^{++}, `cc1` directly emits the expanded code without any optimization at all. When the `gcc` compiler driver invokes the "assembler", we actually have it invoke a program called `gccas` which parses the LLVM assembly file, runs a series of LLVM optimizers on it, then emits a compressed bytecode file (the `.o` file). The interface to `gccas` is intentionally designed to be identical to the interface of the standard system `as` tool, to avoid having to make changes to spec files.

When the user (or a makefile) links the program using our `gcc` compiler driver, it invokes our `gccld` tool. This tool reads the `.o` files specified, links in the appropriate bytecode files from any `.a` files, and then runs a series of interprocedural optimizations on the program. At this time, we directly emit an LLVM bytecode file for the entire program, instead of automatically invoking a native code generator.

Once the program has been optimized and is available in a single bytecode file, there are several ways to execute the resultant program. LLVM provides a very slow (but portable) reference interpreter for bytecode files, a Sparc V9 native code generator, a C back-end, and a Just-In-Time (JIT) compiler for the IA32 architecture.

A large number of LLVM optimizations and analyses are available, including passes for:

- Traditional SSA based optimizations: ADCE, GCSE, LICM, PRE, SCCP, induction variable canonicalization, reassociation, value numbering, register promotion, etc...
- Control Flow Graph based optimizations and analyses: critical edge elimination, loop canonicalization, various dominator, post-dominator, and control dependence graph related analyses, interval construction, natural loop construction, CFG simplification, path profiling instrumentation, etc...
- Interprocedural analyses and transformations: call graph construction, several interprocedural alias analyses, global variable merging, dead global elimination, inlining, Data Structure Analysis [13], automatic pool allocation [12], interprocedural mod/ref, etc...

In addition to pure infrastructure, the LLVM system also provides a large test suite. The three main sections of the test suite are the regression tests (which contain thousands of tests for transformations and other tools), feature tests (which demonstrate how instructions and idioms are used in LLVM), and program tests (which compile benchmarks and other programs with the various code generators, ensuring that they produce code whose behavior agrees with a native compiler). The LLVM web site also hosts a variety of documentation describing aspects of the infrastructure.

LLVM is also still under development. In particular, the C^{++} front-end is nearing completion (runtime library support for exception handling is the major missing portion), Sparc V9 support for the JIT is in development, and a system for runtime optimization of statically compiled binaries is in the research phases.

4.2 Differences from the Proposal

The biggest difference between the proposal and the LLVM implementation is the lack of an LLVM to RTL conversion pass. For our research purposes, we use a C back-end, which provides much of the same functionality as a full fledges RTL back-end, but is much slower. We expect that this component can be added upon demand.

Another big difference between the current implementation and the proposal is the interface between the cc1 program and the mid-level optimizer. For expediency of implementation we currently have the two tools as separate executables, although this obviously incurs more overhead than linking the two components together. Once the subject of including C^{++} code in GCC is better decided, we can look to resolve this issue.

4.3 Support for Developers

One of the strengths of the LLVM infrastructure is that it has some interesting utilities for constructing passes, finding bugs in those passes, and building a compiler around a selection of these passes. This strength is important for two reasons: it allows new people to get into the system and get productive relatively fast, and it also allows experienced developers to be more productive than they otherwise would. The most important features are: a strong consistency checker, a "pass manager", and a tool we call "bugpoint".

The LLVM infrastructure includes a stringent checker for LLVM code, which ensures that type relationships, SSA properties (e.g., all definitions dominates their uses), and other LLVM invariants haven't been violated by a transformation. This checker is automatically run after passes when in development mode to ensure that these passes are not corrupting the input for other passes that are run. Additionally, when in development mode, an automated memory leak detector is automatically enabled, which detects violations of the LLVM representation's ownership model. This light-weight checker is implemented using only a few additions to constructors and destructors for the classes which make up the representation, no garbage collector is necessary.

The LLVM "Pass Manager" provides a structured environment for passes to execute in. Transformations in LLVM use a declarative syntax to indicate which other passes are prerequisites (e.g. break-critical-edges), which analyses are required (e.g. natural loop information, alias analysis, value numbering, interprocedural mod/ref info, etc...), and which analyses are preserved or destroyed by the transformation being run. This structured pass model makes it easier for developers to fit code into the system, and it also makes construction of tools (e.g. gccas and gccld) a simple matter of handling command-line arguments and selecting a sequence of passes to run.

bugpoint, another useful tool, is best described as an "automated test-case reducer". Given an LLVM program (or fragment) and a list of passes to run, it attempts to reduce the test-case (and list of passes) to the minimum which still exposes a problem. bugpoint can currently diagnose passes which crash/assert during optimization and passes which misoptimize the program (by executing the resultant program with a code generator, assuming a deterministic program)[6]. If a test-case causes a pass to crash, bugpoint is usually able to reduce the test-case down to the few LLVM instructions and basic block which cause the problem. If a pass (or combination of passes) miscompiles the test-case, it can isolate a single function which is being miscompiled. The bugpoint tool is possible because of the modularity of the pass manager and the ability to read, write, and modify a representation of whole programs.

4.4 Surprises and Insights from LLVM

Through the experience of developing LLVM, we have developed several insights which may be useful to a broad audience. First, implementing a type-safe linker for C is a non-trivial exercise. C programs often rely on implicit prototypes for called functions, or use prototypes that are blatantly wrong. We have also seen cases where global data is declared to have different types in different translation units (which, in practice, behaves similarly to a COMMON block in FORTRAN). A normal binary linker does not typically have problems with these issues, but they must be handled explicitly with a type-safe linker. On the other hand, this information is often useful to the programmer, like the "lint" tool.

When performing interprocedural analysis, having as much of the program available as possible increases the precision of the analyses. For this reason, we have compiled several libraries to LLVM form that allow them to be analyzed and optimized with the program. This has several interesting consequences: first, the library code itself can be specialized and optimized with the program (for example, optimizing qsort by inlining the comparison functions, so indirect calls do not need to be used). Second, this dramatically reduces the need for adhoc annotations on functions indicating properties

[6] A third mode, for debugging back-end bugs, is planned.

Source Filename	`wc -l` LOC	GCC CSE 1	LLVM Pass Times IC	GER	GCSE	Sum	# LLVM Pass xforms IC	GER	GCSE
combine.c	11103	0.70s	.431s	.027s	.141s	.599s	16182	141	2734
expr.c	10747	0.52s	.141s	.009s	.072s	.222s	6540	41	2870
cse.c	8779	0.50s	.187s	.012s	.061s	.260s	10925	59	1894
reload1.c	7117	0.37s	.058s	.008s	.034s	.100s	5735	86	1830
c-decl.c	6968	0.42s	.022s	.005s	.031s	.058s	3299	3	2221
insn-recog.c	6957	0.34s	.082s	.004s	.090s	.176s	5238	0	654
loop.c	6648	0.33s	.013s	.001s	.003s	.017s	1671	7	264
c-typeck.c	6604	0.46s	.028s	.005s	.026s	.059s	4481	14	1993

Table 1: Transformation timings for source files from the SPEC CPU2000 176.gcc benchmark

such as "`const`" and "`pure`". Instead, simple interprocedural analyses can be used, which have the advantage of applying to user code as well as the built-in functions.

Finally, we have found that investing in making the system easier to develop for, and debug in, has been worth it. In particular, the `bugpoint` tool can narrow down a test-case from thousands of lines of C code to a dozen lines of LLVM code in a few seconds: doing the same manually would take *much* longer. Making the development environment detect problems early is also extremely valuable to developers, making them more productive and making it easier to bring new people on. Having a modular system also helps keep people from getting overwhelmed when they first start on the project.

4.5 Optimizer Performance

The LLVM representation allows for efficient transformations and analyses, both for aggressive interprocedural transformation and traditional optimizations. In order to quantify this performance, we compared the performance of the GCC "`cse`" pass with the performance of the LLVM transformations closest to it (see Table 1). For these tests, we compiled the 8 largest single `.c` files in the SPEC CPU2000 176.gcc benchmark (which is based on the GCC 2.7.2.2 source code). The numbers were collected on a 1.7GHz AMD 2100+ Athlon processor.

The timings for the `cse` pass were collected when compiling with GCC 3.2 and the `-O3` option. The actual timings were acquired as the average of 5 runs with the `-ftime-report` option and the compiler configured for a `i686-pc-linux-gnu` target. The `cse 2` pass was ignored, the timings just include the first invocation of the `cse` pass.

For the LLVM timings, we chose to use a combination of the **I**nstruction **C**ombining, **G**lobal **E**xpression **R**eassociation, and **G**local **C**ommon **S**ubexpression **E**limination passes. The combination of these three phases is believed to be strictly more powerful than the `cse` pass. The Instruction Combining pass supersumes value numbering, constant folding and trivial dead code elimination phases, plus it performs a variety of transformations similar to the GCC "combine" pass (described below). The reassociation pass transforms chained occurrences of commutative operations to promote better code motion. The GCSE pass is a well known technique to remove common subexpressions. The table shows the execution time for each pass as well as the sum of the three. The table also shows the number of transformations that each pass makes (instructions combined, instructions reassociated, common subexpressions deleted).

From the table, we can see that the LLVM optimizations always run in less time than the `cse` pass, and with the exception of the "combine.c" case, took about half as much time. Despite being faster overall, the LLVM transformations are more powerful than the `cse` pass, which only operates on extended basic blocks. The slowest individual transformation by far is the instruction combination pass, which uses a work-list driven approach to perform "peephole" style optimization on the SSA graph (giving it global transformation powers) for a large collection of algebraic identities (such as folding "$(A-(A\&B))$" into "$(A\& \sim B)$"), that the `cse` pass does not perform. Together, the three transformations are quite effective.

In addition to simple scalar optimizations, LLVM is designed to support aggressive interprocedural analyses and optimizations at link-time. As an example, we consider the Data Structure Analysis algorithm, a context-sensitive flow-insensitive memory analysis framework. On the same hardware as above it is capable of analyzing entire programs in seconds: 2.5s the `povray` and 1.2s for the `255.vortex` programs, which are about 136,000 and 67,000 lines of C code respectively [13]. Other simpler algorithms may obviously run much more quickly.

5 Related Work

There is a vast amount of related work on interprocedural optimization in research and commercial compilers [1, 8, 2, 9, 3]. To avoid major changes to the build process, all of these compilers combine the program together at link-time in a very high-level representation, before any substantial optimization is performed. Most often, this representation takes the form of the source language Abstract Syntax Tree (AST) with source language-specific nodes removed. Once the program is combined at link-time, optimization for the entire program commences, starting with interprocedural optimizations.

In contrast, the approach described here immediately optimizes and translates the program to a low-level, but strongly-typed, intermediate representation which is suitable for optimization both at compile- and link-time. Because substantial optimization is performed at compile-time, the interprocedural optimizers have less work to perform at link-time, reducing the amount of time a recompilation requires. Previous work [13, 7, 10, 12] has shown that a low-level representation with type information can support aggressive high-level analyses and transformations.

Another successful class of interprocedural optimizers target very low-level optimizations. These "smart-linkers" typically operate at the level of the machine code, performing optimizations such as interprocedural register allocation and code layout optimizations [16, 14, 5]. Although these tools have been successful, and require little or no modification to the source compiler, they are not capable of performing high-level optimizations at all. Also, these optimizations can all be performed in our framework, because code generation occurs for the entire program at a time, exposing the necessary interprocedural information.

Within the GCC project, several projects in development or recently merged onto the mainline are relevant. In particular, the `ast-optimizer` project and its `tree-ssa` subproject aim to improve optimization in GCC by migrating optimizations from the target-specific RTL representation to a target-independent AST representation. The representation proposed in this paper is similar to the `tree-ssa` GIMPLE representation in some ways (both are language-independent, SSA based, and do not allow nested expressions), but they are different in many other ways.

In particular, the GIMPLE representation is not capable of representing the entire translation unit being compiled: a lot of information about the program is stored only in global variables, or are immediately emitted to the output assembly file. Also, the GIMPLE representation has operations which are closer to the source level. For example, variable definitions can have their address taken, which makes the def-use chain representation much more complex in the GIMPLE representation. On the other hand, the `tree-ssa` project is much better integrated into GCC, is written in the C language, and does not require the introduction of a completely new intermediate representation.

6 Conclusion

This paper presents the design for an aggressive, but realistic, interprocedural optimization component for the GNU Compiler Collection. This design is capable of supporting a broad range of whole-program optimization techniques, is reasonable in terms of compilation time, and has already been implemented. We hope our efforts will accelerate the process of making GCC produce code which is more competitive with commercial compilers, and perhaps LLVM can be directly adopted as an optional part of the compiler itself. We encourage members of the community who are interested in the proposed architecture or LLVM itself to contact the authors with any feedback, questions, or ideas.

References

[1] J. Amaral, G. Gao, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure: A tu-

torial. The International Conference on Parallel Architeture and Compilation Techniques (PACT2000), Oct. 2000.

[2] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. In *Proc. SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pages 301–312, Montreal, June 1998.

[3] D. Blickstein, P. Craig, C. Davidson, N. Faiman, K. Glossop, R. G. S. Hobbs, and W. Noyce. The gem optimizing compiler system. *Digital Technical Journal*, 4(4):121–136, 1992.

[4] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):367–399, 1993.

[5] R. Cohn, D. Goodwin, and P. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), 1997.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 13(4):451–490, October 1991.

[7] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. 2003 ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems*, Feb 2003.

[8] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, (Q4), 1999.

[9] A. Holler and Hewlett-Packard Company. Compiler optimizations for the PA-8000. In *Proc. IEEE International Computer Conference*, 1997.

[10] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. See `http://llvm.cs.uiuc.edu`.

[11] C. Lattner and V. Adve. LLVM Assembly language reference manual, `http://llvm.cs.uiuc.edu/docs/langref.html`.

[12] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[13] C. Lattner and V. Adve. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[14] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992.

[15] R. Stallman. *The GNU C compiler*. Free Software Foundation, 1991.

[16] D. Wall. Global register allocation at link-time. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, 1986.