Link-Time Optimization in GCC: Requirements and High-Level Design

November 16, 2005

1 Introduction

Popular programming languages, such as C, C++, and Fortran, use "separate compilation" to facilitate building large programs. The program is written in various parts, which are usually stored as separate files in the filesystem. Each part is compiled in isolation. The linker combines the separately compiled parts into the executable image.

Although separate compilation is unarguably a useful technique, it has the disadvantage that the compiler is unable to perform optimizations that rely on knowledge about more than one part of the program. Therefore, many modern compilers perform additional optimizations when the program (or a portion of the program) is linked. At link-time, these compilers perform optimizations that would be impossible given any single file. For example, functions can be inlined across files, and optimizations like dead-code elimination, constant-folding, and global data allocation can be performed on all of the program parts simultaneously. Most of the optimizations performed are already performed when compiling a single file; therefore, it is in general possible to reuse much of the infrastructure already present in the compiler.

At present, GCC contains a simple form of this functionality which relies on compiling multiple source files simultaneously. This mechanism is unsatisfactory if the various source files that make up a program are not all available at once (as is true in many large programs), are written in different languages, or need to be compiled with different compiler options. Furthermore, the current functionality is provided only for C programs, and will not be easy to extend to C++ or Fortran programs.

We believe that in order to provide link-time optimization competitive with that provided by other compilers, we must adopt the same technique that they use. In particular, the compiler must save data about each program part during compilation, and at link-time, this data must be read back into the compiler so that the compiler can perform optimizations across multiple program parts.

This document is both an informal requirements document and an informal design document. It provides high-level requirements that we think must be met

by any implementation of link-time optimization. This document also provides a sketch of the implementation approach we intend to use.

The next section explains the semantics of the link-time optimizer, i.e., the way in which it must behave. Section 3 provides a high-level description of the modifications required to the toolchain to support the link-time optimizer. Section 4 sketches the on-disk representation of program parts that will be emitted during compilation, and retrieved during link-time optimization.

2 Semantics

This section explains the semantics that must be observed by the link-time optimizer. The first subsection discusses the external interface for the link-time optimizer, including its relationship to the compiler driver. The second subsection explains how the link-time optimizer will combine multiple translation units into a single unit.

2.1 External Interface

Requirement 1 *Use of link-time optimization must require only the addition of a single command-line option at compile-time and link-time. The link-time optimizer must not require access to input source files, or any other data other than that embedded in the object files by the compiler.*

Note: The compiler could choose to embed the entire source file in the object file without violating this requirement. The point of this requirement is not to restrict the kind of information which the compiler embeds in the relocatable object file; rather, the point is that the inputs to the link time optimizer should be the same relocatable object files the user would have at hand if link-time optimization were not in use. – *End note.*

Rationale: The objective of this requirement is ease of use. Users should not have to maintain "on-the-side" databases, or otherwise modify the workflow to which they are accustomed. When link-time optimization is in use, the compiler should still generate assembly files, the assembler should still generate object files, and the linker should still combine those object files. Any other paradigm will require overly invasive changes on the part of users, which will be a significant barrier to adoption. The paradigm suggested here will require only the addition of a single option to the `CFLAGS` make variable to enable link-time optimization in many programs. – *End rationale.*

Requirement 2 *The link time optimizer must be able to operate on a subset of the complete program. The input to the link-time optimization step will be one or more relocatable files (e.g., ELF `ET_REL` files) . The output from this step will be a single relocatable file containing the (optimized) contents of the input files.*

Rationale: Whole-program optimization is the specific case of link-time optimization in which it is assumed that the optimizer is able to see the entire program. That assumption is useful in that, for example, functions that are unreferenced can be discarded, even if they have external linkage. However, limiting the link-time optimizer to the whole-program case would preclude link-time optimization in common cases. For example, the use of a single hand-written assembly code module might prevent link-time optimization. Or, it might be impossible to perform link-time optimization when creating a shared library.

Therefore, while a whole-program mode would indeed be useful, it is essential that the link-time optimizer be able to operate on a subset of the object files that will make up the eventual program. – *End rationale.*

Requirement 3 *The set of symbols with global or weak binding (in ELF, any symbol with an* `ELFXX_ST_BIND` *value of* `STB_GLOBAL` *or* `STB_WEAK`*) defined in the output relocatable file must be the union of such symbols in the input relocatable files.*

Requirement 4 *Those entities with non-global, non-weak binding in the input files which are present in the output file must have the same names in the output file that they have in the input file.*

Rationale: It is valid to combine two C translation units which both define a function with internal linkage named `f`. In order to support debugging of optimized programs, these functions must continue to be named `f` in the output file. – *End rationale.*

Requirement 5 *The link-time optimizer must take one of the following actions:*

- *Combine the input relocatable files into an output relocatable file.*
- *Issue an error message indicating that the combination was invalid, i.e., that any program containing this combination would be invalid, according to the relevant language standards.*
- *Refuse to the perform the combination, despite the fact that the combination is valid.*

Note: A trivial implementation of Requirement 3 would be to use the traditional linker, invoked with the `-r` option, as the link-time "optimizer". This "optimizer" would only make use of the first two options above. The intent of the requirement is that the driver, when invoked to perform link-time optimization, will behave as a "smart" drop-in replacement for `ld -r`. – *End note.*

Requirement 6 *If the link-time optimizer refuses to perform a valid combination, the compiler driver must run* `ld -r` *(or an appropriate system equivalent) to combine the object files.*

Rationale: Certain combinations of object files may be permitted by the language standards, but may require undue effort to implement in the link-time optimizer. In addition, the level of support (if any) for link-time optimization may vary from architecture to architecture or from system to system. In these cases, rather than issue an error, which would disrupt the build compilation process, the compiler driver will invoke `ld -r` (so that the build process proceeds. This mechanism will make it possible for users to incorporate use of the link-time optimizer in their build processes without needing to conditionalize that use. – *End rationale.*

Requirement 7 *If command-line options used to compile the input relocatable object files are incompatible, the link-time optimizer must refuse to perform the combination.*

Rationale: The use of GCC command-line options (such as `-ftrapv`) are global options that affect the semantics of GIMPLE. Ideally, the impact of these options might be represented directly in GIMPLE; for example, we might have both "trapping addition" and "non-trapping addition" operators, rather than using a global variable to indicate what kind of addition is performed by a single addition operator. However, at present, GCC's internal representation cannot simultaneously describe both operations. Therefore, if one input object file uses command-line options incompatible with the options used to compile another, the link-time optimizer must reject the combination. – *End rationale.*

2.2 Combination of Translation Units

This document describes the semantics for combinations of translation units, assuming that all of the input translation units were written in either the C or C++ programming languages. This simplifying assumption is not meant to indicate that the mechanisms used are intended to be language-specific; rather, it is explicitly our intent that the mechanisms be language-independent, and that it be possible to add support for new programming languages to the existing link-time optimization framework.

However, each additional language will require some amount of thought about what it means to combine code written in that language with the code written in other languages. The key questions that must be answered for each language are:

- Which entities are the same? For example, is type T_1 written in language L_1 the same type as type T_2 written in language L_2?
- If two entities or two types are the same, is the combination valid? For example, two variables with external linkage and the same name in the relocatable object file *must* be the same variable, because any ordinary linker would consider them the same. However, if the variables do not have the same type, then the combination is invalid.

The remainder of this section endeavors to answer those questions for the C and C++ programming languages.

The C and C++ programming language standards are written in terms of individual "translation units" which are then combined. In GCC, each relocatable file is the compiled form a single translation unit. The language standards state that certain combinations of translation units (in order to form a single program) are valid, while others are invalid. This section explains describes the requirements that the link-time optimizer must obey when combining translation units.

For example, it is not valid in C to combine two translation units if both declare a variable with external linkage named `a`, but in one translation unit the variable is given type `int` and in the other translation unit the variable is given type `double`. It is also invalid to combine two translation units that define a function with external linkage named `f`. Of these, only the second invalid combination is diagnosed by most linkers.

Note: As per Requirement 6, the link-time optimizer may choose not to combine translation units, even though such a combination would be valid. The requirements in this section apply only if the link-time optimizer does in fact perform the combination. If it does not perform the combination (and, therefore, relies upon the compiler driver to use the traditional linker to perform the combination), then, of course, these requirements do not apply. – *End note.*

Requirement 8 *Valid combinations of translation units must behave as required by the appropriate language standard(s).*

Requirement 9 *Invalid combinations of translation units which would result in a diagnostic if linked together by the traditional linker should also result in a diagnostic when combined via link-time optimization.*

Note: A trivial implementation of Requirement 9 would be to directly invoke the linker, solely for the purpose of obtaining any diagnostics, before performing link-time optimizations.

These requirements do not preclude diagnosing invalid combinations that would not be diagnosed in the ordinary process of linking. In fact, some programmers may desire these additional diagnostics and welcome the stricter standards conformance implied by such diagnostics. However, experience suggests that many programs contain technically invalid combinations that, in practice, do not result in problems. A common example is the declaration of a variable in one translation unit as `int` while the same variable is declared as `long` in another translation unit. On an ILP32 system, these types, while not technically compatible, are both declarations of a 32-bit integer type. The link-time optimizer need not perform such invalid combinations, but issuing a fatal error may not be the most appropriate action in such a situation. – *End note.*

In order to combine the input translation units, the link-time optimizer must recognize that certain entities in one translation unit are "the same" as entities in another translation unit. Entities that are the same must be "merged" into a

single entity. There are three kinds of entities that must potentially be merged: types, variables, and functions.

Without loss of generality, we consider only two translation units; for expository purposes, we consider the combination of multiple translation units as successive combinations of pairs.

For each kind of entity, we first describe how to determine whether or not two entities are the same. Then, once the entities are determined to be the same, we indicate whether or not the combination is "valid", "invalid", or "difficult". An invalid combination is a combination prohibited by the relevant language standards; such a combination represents an error in the input program. A difficult combination is one which is permitted by the relevant language standards, but in which the combination would be fundamentally difficult to optimize.

Requirement 10 *The link-time optimizer should perform valid combinations and should issue (possibly fatal) errors for invalid combinations. If the link-time optimizer elects not to perform a difficult combination, it must issue a diagnostic explaining the source of confusion.*

2.2.1 Variables and Functions

This section explains how to tell if two variables, or two functions, are the same, and, if they are the same, whether the combination is valid, invalid, or difficult.

Let E_1 be a variable or function from the first translation unit and E_2 be a variable or function from the second translation unit.

If either E_1 or E_2 has a binding other than global or weak, then E_1 and E_2 are not the same.

Otherwise, E_1 and E_2 are the same if (and only if) they have the same name, in the form that such names appear in the relocatable file symbol table. In particular, for languages like C++, in which the names of entities in the source program cannot be directly represented in traditional relocatable file formats, the names used in the relocatable file symbol table are "mangled" by most compilers, including the compilers in the GNU Compiler Collection. It is these mangled names that are used to determine identity.

Note: GCC provides mechanisms for overriding the name used for an entity in a relocatable file, so that, for example, a variable named `i` in the source code may be named `j` in the relocatable file. In that case, it is `j` (not `i`) that is used to determine identity. – *End note.*

The combination is invalid if and of the following conditions hold:

- E_1 is a variable and E_2 is a function, or vice versa.
- E_1 and E_2 do not have the same type, in the sense of Section 2.2.2, except that if the type of one of E_1 or E_2 involves an incomplete array type, this may be replaced with a complete array types in the type of the other entity, so long as the element types are the same.
- Both E_1 and E_2 are definitions, but (a) neither E_1 nor E_2 is weak, and (b) E_1 and E_2 are not both C++ inline functions.

- E_1 and E_2 have incompatible GNU attributes.

The combination is difficult if:

- E_1 and E_2 are functions, and the type of one uses an unprototyped function type, while the other does not.

Note: The C99 programming language permits multiple definitions of inline functions, even if those definitions are not identical. A C99 inline function must always have a canonical, non-inline definition. The semantics are that the address of such a function is always that of the canonical definition. Calls may use either the inline definition provided in the same translation unit as the caller, or the canonical definition, but not an inline definition in some other translation unit. The simplest way to handle this complexity is for the compiler to omit the inline definitions from the intermediate representation, even if it has used them for early inlining by that point. If we elect to emit the inline definitions as well, then combinations involving at least one C99 inline definition should probably be considered difficult. – *End note.*

Otherwise, the combination is valid.

If either E_1 or E_2 is a strong definition, that definition is used. Otherwise, if exactly one of E_1 and E_2 are definitions, then that definition is used. Otherwise both E_1 and E_2 are declarations, or both are weak definitions, or both are C++ inline definitions; in that case, the link-time optimizer may use either E_1 or E_2.

2.2.2 Types

This section explains how to tell if two types are the same, and, if they are the same, whether the combination is valid, invalid, or difficult.

Note: Types, in contrast to variables and functions, are not combined by traditional linkers. Therefore, types present a problem that is less well-understood than the corresponding problem for variables and functions. – *End note.*

Types with differing cv-qualifiers are never the same. In the absence of GNU attributes, corresponding integral types, corresponding floating point types, and the `void` type are the same in all translation units. Array types are the same type if their element types are the same and if the number of elements in the array is the same. Function types are the same type if the return type and arguments types are the same. Pointer (reference) types are the same if the pointed-to (referred-to) types are the same. Pointers to non-static members are the same if the pointed-to types and class types are the same. In all of these cases, if GNU attributes are present, types which would be the same be it not for their GNU attributes may in fact be different due to the use of GNU attributes.

Note: The preceding rules imply that type-equivalence for these basic types is "structural", i.e., does not depend on the name of the types. The reason for this rule is that C and C++ `typedef` names are irrelevant, for the purposes of comparing types. – *End note.*

If T_1 and T_2 have the same name and are both enumeration types, then T_1 and T_2 are the same.

If T_1 and T_2 have the same name (i.e., in C++, would have the same `std::type_info` object) and both are class types[1], then T_1 and T_2 are the same.

Note: The preceding rules imply that type-equivalence for class and enumeration types is not structural; rather, it depends on the names of the types. However, the validity rules below impose a structural consistency on the types that are the same. – *End note.*

Otherwise, the combination is invalid if both types were written in C++ translation units, and at least one of the following conditions hold:

- T_1 and T_2 are both complete enumeration types and do not have the same minimum and maximum values.
- T_1 and T_2 are both complete class types and do not have the same size and alignment.
- T_1 and T_2 are both complete class types do not have non-static data members (including those implicitly generated by the compiler) of the same types at the same offsets, or do not have the same base class types at the same offsets, or do not have the same set of virtual functions, declared in the same order.
- T_1 and T_2 have incompatible GNU attributes.

Note: If two enumeration types have the same minimum and maximum values, they have the same underlying type. – *End note.*

The combination is difficult if at least one type was written in the C programming language, but the combination would be invalid if both types were written in C++.

If either T_1 or T_2 is a complete type, then the complete type is used. Otherwise, the link-time optimizer may select either type.

3 Architecture

The link-time optimizer will not be an entirely separate toolchain component. Instead, the link-time optimizer will reuse technology from the existing in order to provide a shorter time-to-solution, to reduce the effort required to port to new platforms, and to provide a mechanism for sharing optimization capabilities between the ordinary and link-time optimizers.

Of course, many components of the toolchain must be modified to support link-time optimization. The remainder of this section describes, at a high level, the modifications that will be made to the established toolchain architecture.

[1] Class types include those types declared with the `class`, `struct`, and `union` keywords

3.1 Emission of Information

The compiler will be modified to emit new special sections containing the stack machine representation of the tree structure. (For details on the format of the representation, see Section 4.2.) The compiler will also be modified to emit additional DWARF-3 sections, as necessary, to describe variables in function bodies. The stack code will be emitted using numeric literals so that no assembler support will be required.

Note: Systems that do not support special sections may store the information required using some other mechanism. – *End note.*

3.2 Link-Time Optimizer Front End

A new GCC front end will be provided to serve as the link-time optimizer. The input language for this front-end will be relocatable object files, not programming language source code. The front end will extract the relevant data from the relocatable object files, build a TREE representation for the input program, and, then, use the same optimizers and back-ends already present in GCC.

In order to accommodate Requirement 4, the back end will be modified to permit the generation of multiple symbols with the same name, if those symbols have internal linkage. This modification will be conditional on appropriate assembler support.

3.3 Assembler

The assembler must be modified to permit multiple symbols with the same name, if at most one such symbol has external linkage. Obviously, there must be some way of indicating to which of these symbols a particular reference applies. The simplest solution to this problem is to continue to use unique names for the symbols throughout the majority of the assembly file, but to provide a mapping table indicating how symbols should be renamed before they are emitted in the relocatable file.

3.4 Linker

The linker contains logic for determining which set of objects should be extracted from archives (typically, files with the `.a` extension) when performing a link. The linker will be modified to provide a mode in which this information is emitted, but no actual linking is performed. This facility will allow the link-time optimizer to work only with relocatable files and avoid duplication of code already present in the linker.

3.5 Driver

The driver will be modified to invoke the link-time optimizer front end at link time, when an appropriate option is provided on the command-line. If the link-time optimizer successfully produces an assembly file, the driver will then invoke

the assembler, and thereby generate a new object file. If the link-time optimizer exits with a fatal error, the driver will exit with a non-zero exit code. Finally, if the link-time optimizer indicates that it is unable to optimize the input files, the driver will invoke the linker, with the `-r` option, to perform an unoptimized partial link. In this last case, the link-time optimizer will have already issued a diagnostic as per Requirement 10.

The driver will also provide a mode where the partial link performed by the link time optimizer is followed by a full link. Using this mode, it will be possible to take advantage of link-time optimization simply by providing one additional option at link time.

4 Representation

Link-time optimization requires that the compiler emit some information during initial compilation. This information will be read back in by the link-time optimizer. This section discusses the representation of that information.

Requirement 11 *The representation used must be "compiler-independent" in the sense that it should be feasible for tools other than GCC (and written in languages other than C) to read the representation generated.*

Rationale: The simplest way to represent the information GCC needs would be to serialize the TREE data structures to disk. Since these are the data structures GCC uses for optimization and code generation, we know that these data structures contain the information required. However, there are a number of disadvantages to this approach.

First, different versions of the compiler will almost certainly be unable to interoperate. Minor differences in the tree structure from version-to-version of the compiler are inevitable. In fact, different builds of the same version of the compiler may not have precisely compatible tree structures; for example, for some host/target combinations, either 32-bit or 64-bit `HOST_WIDE_INT`s may be used, but the resulting in-memory tree representations are different. It is highly desirable to support link-time optimization of relocatable files built by one provider on the systems of another provider, and there is no guarantee that both environments are necessarily using the same version of the compiler.

Second, there are a large number of additional tools that would benefit from being able to access the information possessed by the compiler in the middle of its processing. Lint-like source analysis tools, IDEs, and other similar tools are all potential clients. Various projects to generate XML from GCC have been attempted in order to try to facilitate this kind of usage. – *End rationale.*

It might be possible to use a well-known representation format, such as Java bytecode or Microsoft's CIL. However, in addition to concerns about possible intellectual property issues and alignment with particular vendors, we also concluded that neither of these formats was well-suited to GCC's needs. Java bytecode is too Java-specific for a multi-language compiler. CIL is too closely

tied to Microsoft's virtual machine architecture. Therefore, we concluded that it is necessary to develop a new representation format.

Requirement 12 *The representation format should be extensible to contexts other than link-time optimization.*

Rationale: The primary purpose of the representation format is for link-time optimization. However, while the link-time optimizer may want to process a language-neutral representation of the program that has been partially optimized before emission, other tools (such as the IDEs and source-analysis tools mentioned above) will likely want to access an unoptimized representation of the program that contains more information about the way in which the user originally formulated the program. In particular, an implementation of the C++ "export" keyword will want a representation of the program that is C++-specific. GCC's TREE representation is sufficiently flexible to represent these various levels; the on-disk representation should be similarly flexible. – *End rationale.*

Requirement 13 *The semantics of the representation (for each supported level of representation) must be well-specified.*

Rationale: In order to facilitate the interoperability described in Requirement 11, we must have a well-specified explanation of the meaning of the representation. Ideally, we would have an operational semantics capable of formally describing execution of the input program, but that goal is not realistically achievable; instead, we will have to settle for an informal document describing the semantics.

Each level of representation should be documented, as part of the GCC manual, and handled through the normal GCC development process. We do not intend to encourage ad-hoc extensions to the format for the use of particular individual needs. – *End rationale.*

Requirement 14 *The representation used for link-time optimization should correspond approximately to the GIMPLE TREE representation.*

Rationale: The RTL level would be too low-level a representation for link-time optimization as its use would preclude the use of the increasingly powerful TREE optimizers.

GIMPLE is a language-independent representation that has been simplified to make it amenable to optimization. However, high-level information, such as the types of expressions remains.

Before emitting information for link-time optimization, it may be useful to apply existing optimizations that reduce program size, such as constant propagation, dead code elimination, and, perhaps, a limited form of inlining. Functions with internal linkage that are never referenced can be eliminated.

We do not yet have an opinion as to whether the representation should be in SSA form. While the information provided by SSA form is quite useful, it

can be easily regenerated. PHI nodes, and the use of additional variables, may so bloat the representation that it is more efficient to emit a non-SSA form.

The use of general compression algorithms may be undesirable, to the extent that they make decoding the data difficult, including preventing random access to the data. However, it might still be desirable to use such an algorithm, if the compression achieved is sufficiently significant. In addition, some forms of context-sensitive compression may be useful. Almost all externalized byte code representations use a representation that is a variant of three-address code. These representations often require a large number of intermediate variables. Schemes, such as dividing the program into blocks such that a limited number of variables are present in each block, and then using small integers to refer to the registers with a given block, can reduce the cost. – *End rationale.*

Requirement 15 *The representation format must be designed to support forward evolution. For example, the format should permit the addition of new operators as such operators are added to GIMPLE. It should be possible to perform link-time optimization using input object files generated with different versions of the compiler, provided that the link-time optimizer used is that corresponding to the newest compiler used to build the object files.*

Note: This requirement is not meant to imply that an old version of the link-time optimizer should be able to process objects created by a newer version of the compiler. Rather, a new version of the link-time optimizer should be able to process objects created by an older version of the compiler. There may, occasionally, be cases where that is not possible, and the representation format should contain version numbers that permit the link-time optimizer to identify such cases. – *End note.*

Rationale: Since the format is intended for use not only by the link-time optimizer, but also by other tools, the format must be fundamentally stable. However, as new optimizations and new language features are added to GCC, it may occasionally be necessary to add new operations to GIMPLE, new kinds of types, and so forth. Therefore, the format must support forward evolution. – *End rationale.*

4.1 Declarative Representation

This subsection outlines the representation of declarative entities, i.e., types, variables, functions (their declarations, not their bodies), and other similar entities. The bodies of functions are discussed in Section 4.2 below.

The representation for types will be DWARF-3. Although the DWARF-3 standard was originally designed for providing debugging information, it's representation for types provides nearly all of the information required for link-time optimization, including the location of non-static data members and base classes. Furthermore, DWARF-3 specifically provides for vendor extension, allowing us to record any additional information that we might require. In particular, we will extend DWARF-3 to represent relevant GNU attributes.

Note: Description of GNU attributes in DWARF-3 will also be of use to GDB and other debuggers, to the extent that, for example, these attributes affect the calling conventions of functions using these types. While this proposal contemplates the use of the DWARF-3 extensions in service of link-time optimization, there is no reason the same extensions should not be used as part of the conventional debugging information. – *End note.*

DWARF-3 has several other advantages for our purposes, including:

- DWARF-3 is a well-specified standard.
- GCC already knows how to emit DWARF-3 information. Therefore, the effort required to emit the information will be much smaller.
- Various tools, including `readelf`, know how to interpret and display DWARF-3 information. Therefore, it will be possible to more easily verify the generated information, and construction of the portion of the link-time optimizer that will read this information and reconstitute appropriate TREE nodes will be easier.

The representation for declarations will also be DWARF-3. For function declarations and for variables with static storage duration (i.e., "global" variables) this representation is very natural. For automatic variables, the representation is less natural, in that DWARF-3 scope information refers to byte offsets within code sections. In other words, the DWARF-3 representation for a local variable says that it comes into scope at offset N_0 (from the start of a designated section) and goes out of scope at offset N_1. References into the ordinary code sections clearly will not be useful to the link-time optimizer. Therefore, a separate DWARF-3 section will be used which references not the usual code section, but, rather, the section used by the executable representation.

Some variables with static storage duration require initialization. If the initialization should be performed dynamically, no special handling is required; the dynamic initialization will appear in the ordinary code stream. If the initialization is performed statically, the initializer will be recovered directly from the data section in the relocatable file, as there is no convenient representation for initialization in DWARF-3.

4.2 Executable Representation

The executable portion of the intermediate code will be "bytecode" for a stack machine hereinafter known as the GNU Virtual Machine (GVM). The stack machine will represent code at approximately the same level represented by the GIMPLE level of TREE. The use of a stack removes any need for pickled pointers to represent expression trees. The TREE representation of the executable code can be readily reconstituted from the stack code.

The GVM will share some features with Sun's Java Virtual Machine (JVM) and Microsoft's Common Intermediate Language (CIL). However, there will also be some important differences:

portability Like JVM bytecode, the format of the GVM bytecode will be architecture-independent. However, unlike JVM bytecode, it will not be possible to reuse the GVM bytecode from one platform on another; for example, the bytecode generated by an IA32 GNU/Linux will not be reusable as part of a PowerPC AIX program.

executability Given that the GVM code will be translated into executable code, it is highly likely that this code could be directly executed. However, no particular attention will be given to make this easy. What is important is to be able to transport the intermediate code efficiently from the compiler to the link time optimizer.

verification Some bytecode formats permit verification; i.e., a tool can check that executing the bytecode will not result in certain kinds of unsafe behaviors. The Java bytecode format is fully verifiable. The CIL format provides optional verification. The GVM format will not be designed to support verification because most of our target languages explicitly permit operations that would be considered unsafe by a verifier.

operators The operators will be drawn from two sets: those that manipulate the stack (`load`, store, duplicate, swap) and the operators used in GIMPLE.

Note: We believe that it may be useful to add some higher-level operations to GIMPLE, such as array operations, virtual functions calls, dynamic casts, parallel execution blocks, and high level loops. To the extent that these higher-level operators will facilitate additional optimizations, they should be present in GIMPLE, and, therefore, in the stack machine. As GIMPLE evolves, the stack machine should evolve in parallel. – *End note.*

the stack The stack is an unbounded array of expressions. It will be an invariant of the code generated that the stack will be empty at basic block boundaries. Basic block boundaries will be explicitly indicated in the stack machine code.

JVM and CIL codes utilize a register allocation pass to generate their intermediate code. This register allocation obscures the code in a way that makes debugging more difficult and requires significant resources on both the input and output sides. Therefore, the generation of GVM bytecode will not require register allocation.

blocks There will be several types of explicit blocks in the code stream. These are basic blocks, bind expressions, parallel execution blocks.

the registers The registers of the GVM correspond one for one with the local variables at the GIMPLE level. Each local variable will be assigned an index, and a symbol table will be provided to keep the type of each of these temps. Types will be given as references to the corresponding DWARF-3 DIEs representing those types.

types The stack is dynamically typed. The type of an element on the stack is determined from the operation used to set that item. The result type of a generic operation (e.g., addition) is determined from the inputs to that operation. Some operations (e.g., casts) will explicitly indicate the result of the operation.