Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

CLI Back-End in GCC

Roberto Costa
STMicroelectronics

Andrea C. Ornstein
STMicroelectronics
andrea.ornstein@st.com

Erven Rohou
STMicroelectronics
erven.rohou@st.com

Abstract

CLI is a framework that defines a platform independent format for executables. The framework has been standardized by ECMA and ISO; CLI is also known as the foundation of .NET framework.

In June 2006, a project aimed at the development of a back-end producing CLI-compliant binaries was born in GCC community. One year later, the back-end supports C99 (with a few exceptions) and it already delivers excellent results, both in terms of performance and code size.

This paper presents the internal structure of CLI back-end and its current status.

1 Introduction

CLI (*Common Language Infrastructure*) is a framework that defines a platform independent format for executables and a run-time environment for the execution of applications. CLI was invented and it is still best known to be the foundation of Microsoft .NET framework.

CLI executables are encoded in a *Common Intermediate Language* (*CIL*), a machine-independent instruction set. This is possible since CIL is not bound to the instruction set of the machine on which applications are executed. Since a CLI application does not contain native code, it is not directly executable. A CLI virtual machine is required in order to run a CLI binary; a wide range of execution techniques are possible, which include interpretation, ahead-of-time and just-in-time compilation.

CLI is standardized by the European Computer Manufacturers Association (ECMA) [5] and by the International Organization for Standardization (ISO) [9]. Following the standardization, CLI is gaining momentum in the open source community. Two advanced open-source implementations are Mono [3] and Portable.NET [1].

In order to fill the gap of a performing C/C++ open compiler, the project for a CLI back-end[1] in GCC was born in June 2006, led by a small group of people working at STMicroelectronics, in Manno (Switzerland) research lab. The purpose of the project is to develop a GCC back-end that produces CLI-compliant binaries. The initial focus is on C language (more precisely, C99); C++ is likely to be considered in the future, as well as any other language supported by GCC for which there is an interest for a CLI back-end. The implementation currently resides in `st/cli` branch of GCC subversion repository.

About one year later, this papers presents the internal structure of CLI back-end and it gives a detailed description of its specific passes. Finally, the status of the back-end is explained, along with the current areas of work.

2 Related work

CLI back-end in GCC is not the first attempt of a C compiler producing CLI-compliant binaries.

Microsoft Visual Studio was the first development environment to support CLI binaries; such support was introduced in Visual Studio .NET 2002 and improved in the following revisions (Visual Studio .NET 2003 and Visual Studio 2005). Microsoft Visual Studio provides a C++ compiler, with additional features that give developers full access to managed features of CLI framework as well as traditional unmanaged C/C++ code. 2002 and 2003 editions referred to these as *Managed C++ extensions*; in Visual Studio 2005, they were superseded by *C++/CLI extensions*, standardized by ECMA [4].

Microsoft Visual Studio is proprietary software and, in particular, it does not grant any of the four kinds of freedom specified in GNU definition of free software [6].

[1] `http://gcc.gnu.org/projects/cli.html`

A project for a C-to-CLI compiler sprang out of Microsoft research is the .NET back-end for *lcc*, better known as *lcc.NET* [8]. As stated by the project authors, *lcc* port to CIL is a "realistic test of how well CIL supports" C language. The outcome of the project should be considered a working prototype more than a production compiler; in addition, *lcc* license does not comply with GNU definition of free software either and, to the best of our knowledge, *lcc.NET* is not actively maintained nor improved. Nevertheless, *lcc.NET* experience confirms that C is truly within the range of languages that CLI is meant to support.

Portable.NET [1] project includes *cscc*, a C (and C#) compiler covered by GPLv2 license. *cscc* focuses on complete support of C language, but it lacks many optimizations; this makes the quality of the bytecode it generates not very good.

CLI back-end is also not the first attempt to support a bytecode-based virtual machine as a GCC target. As a matter of fact, *gcj* [7] project includes a GCC back-end for the Java virtual machine. gcj does much more than that, since it supports compilation of Java source code to Java bytecode (class files) or directly to native machine code, and Java bytecode to native machine code. In other words, gcj includes a front-end for the Java programming language, a front-end and a back-end for the Java bytecode.

Java bytecode is the closest thing to CIL bytecode already supported in GCC; hence, gcj has been of inspiration for the CLI back-end project. Similarly to gcj, in the future the scope of CLI support in GCC may be enlarged up to include a CLI front-end and a run-time library able to execute the bytecode itself.

Following this direction, a preliminary CLI front-end has been recently developed by Ricardo Fernández Pascual at STMicroelectronics (research funded by HiPEAC network of excellence [2]). In its current status, the front-end focuses on the CIL opcodes which are currently generated by the CLI back-end; therefore, object-oriented bytecodes are not supported. This work is likely to be integrated soon into CLI support of GCC.

3 Structure of the back-end

Unlike a typical GCC back-end, CLI back-end stops the compilation flow at the end of the middle-end passes and, without going through any RTL pass, it emits CIL bytecode from GIMPLE representation. As a matter of fact, RTL is not a convenient representation to emit CLI code, while GIMPLE is much more suited for this purpose.

CIL bytecode is much more high-level than a processor machine code. For instance, there is no such a concept of registers or of frame stack; instructions operate on an unbound set of locals (which closely match the concept of local variables) and on elements on top of an evaluation stack. In addition, CIL bytecode is strongly typed and it requires high-level data type information that is not preserved across RTL.

3.1 Target machine model

Like existing GCC back-ends, CLI is truly seen as a target machine and, as such, it follows GCC policy about the organization of the back-end specific files.

Unfortunately, it is not feasible to define a single CLI target machine. The reason is that, in dealing with languages with unmanaged data like C and C++, the size of pointers of the target machine must be known at compile time. Therefore, separate 32-bit and 64-bit CLI targets are defined, namely `cil32` and `cil64`. CLI binaries compiled for `cil32` are not guaranteed to work on 64-bit machines and vice-versa. Current work is focusing on `cil32` target, but the differences between the two are minimal.

Being `cil32` the target machine, the machine model description is located in files `config/cil32/cil32.*`. This is an overview of such a description:

- The size of pointers is set to 32 (this is `cil32` target, it would similarly set to 64 for `cil64`). Natural modes for computations go up to 64 bits.
- Alignment rules specify that natural alignment is always followed (more precisely, in the absence of `packed` attribute).
- Properties exclusively needed by RTL passes are skipped. This is a mere consequence of the fact that CLI back-end starts from GIMPLE and it does not go through RTL at all.
- Though CLI back-end does not reach RTL passes, there is a minimum set of RTL-related description

that must be present anyway. For instance, a few instruction selection patterns are mandatory, while others are used by some heuristics for cost estimation; there must be a definition of the register sets and a few peculiar registers have to be defined... As a rule of thumb, the machine model contains the simplest description for these properties, even if this makes little sense for CLI target.

3.2 CIL simplification pass

Though most GIMPLE tree codes closely match what is representable in CIL, some simply do not. Those codes could still be expressed in CIL bytecodes by a CIL-emission pass; however, it would be much more difficult and complicated to perform the required transformations at CIL emission time (i.e.: those that involve generating new local temporary variables, modifications in the control-flow graph or in types...), than directly on GIMPLE expressions.

Pass `simpcil` (file `config/cil32/tree-simp-cil.c`) is in charge of performing such transformations. The input is any code in GIMPLE form; the outcome is still valid GIMPLE, it just contains only constructs for which the CIL emission is straightforward. Such a constrained GIMPLE format is referred as "CIL simplified" GIMPLE throughout this documentation.

The pass is currently performed just once, after leaving SSA form and immediately before the CIL emission. This is not a constraint; the only requirement is that the CIL emission is immediately preceded by a run of `simpcil`. `simpcil` pass is designed to be idempotent and it is perfectly fine to insert additional previous runs in the compilation flow. Given its current position in the list of passes, `simpcil` does not yet support SSA form (though planned).

This is a non-exhaustive list of `simpcil` transformations:

- Removal of `RESULT_DECL` nodes. CIL does not treat the value returned by a function in any special way: if it has to be temporarily stored, this must happen in a local. A new local variable is generated and each `RESULT_DECL` node is transformed into a `VAR_DECL` of that variable.
- Expansion of `LROTATE_EXPR` and `RROTATE_EXPR` nodes. In CIL there no are opcodes for rotation and they have to be emulated through shifts and bit operations. A previous expansion may generate better code (i.e.: it may fold constants) or trigger further optimizations.
- Expansion of `ABS_EXPR` nodes (in case of the `-mexpand-abs` option), of `MAX_EXPR` and `MIN_EXPR` nodes (in case of the `-mexpand-minmax` option) and of `COND_EXPR` nodes used as expressions (not statements). The expansion requires changes to the control-flow graph.
- Expansion of `LTGT_EXPR`, `UNEQ_EXPR`, `UNLE_EXPR`, and `UNGE_EXPR` nodes. The CIL instruction set has some support for comparisons, but it is not orthogonal. Whenever a comparison is difficult to be translated in CIL, it is expanded.
- Expansion of `SWITCH_EXPR`, when it is not profitable to have a switch table (heuristic decision is based on case density). The CIL emission pass always emits a `SWITCH_EXPR` to a CIL switch opcode. When a low case density makes compare trees preferable, the `SWITCH_EXPR` is expanded; otherwise the `SWITCH_EXPR` is not modified. The expansion requires changes to the control-flow graph.
- Expansion of `COMPONENT_REF` nodes operating on bit-fields and of `BIT_FIELD_REF` nodes. CIL has no direct support for bit-field access; hence, equivalent code that extracts the bit pattern and applies the appropriate bit mask is generated. Memory access is performed by using `INDIRECT_REF` nodes. Beware that such nodes on the left-hand side of an assignment also requires a load from memory; from the memory access point of view, the operation cannot be made atomic.
- Expansion of `TARGET_MEM_REF` nodes. Emission of such nodes is not difficult; however, a previous expansion may trigger further optimizations (since there is no similar construct in CIL bytecodes).
- Expansion of `ARRAY_REF` nodes with non-zero indexes into `ARRAY_REF` with zero indexes. CLI supports managed arrays, but their semantics is not appropriate for C-style arrays; CLI arrays are

managed data structures (objects), with all the consequences deriving from that (managed memory, garbage allocation, automatic bound checking). Therefore, `ARRAY_REF` nodes cannot be directly translated into CLI arrays. The CIL emission of such nodes is not difficult; however, a previous expansion may generate better code (i.e.: it may fold constants) or trigger further optimizations. Remark that such a simplification must keep `ARRAY_REF`s, they cannot be replaced by `INDIRECT_REF` nodes in order not to break strict aliasing.

- Expansion of `CONSTRUCTOR` nodes used as right-hand sides of `INIT_EXPR` and `MODIFY_EXPR` nodes. Such `CONSTRUCTOR` nodes must be implemented in CIL bytecode through a sequence of finer grain initializations. Hence, initializer statements containing `CONSTRUCTOR` nodes are expanded into an equivalent list of initializer statements, with no more `CONSTRUCTOR` nodes.

- Rename of inlined variables to unique names. Emitted variables keep the original name. In case of variables declared within inlined functions, renaming them is needed to avoid clashes.

- Globalization of function static variables. CIL locals can be used for function non-static variables; there is no CIL feature to do the same with function static variables. Therefore, those variables have their scope changed (they become global), and their name as well, to avoid clashes.

- Expansion of initializers of local variables. In order to simplify the emission pass, the initialization of local variables (for those that have it) is expanded into the body of the entry basic block of the function.

3.3 CIL emission pass

Pass `cil` (file `config/cil32/gen-cil.c`) receives a CIL-simplified GIMPLE form as input and it produces a CLI assembly file as output. It is the final pass of the compilation flow.

Before the proper emission, `cil` currently merges GIMPLE expressions in the attempt to eliminate local variables. The elimination of such variables has positive effects on the generated code, both on performance and code size (each of such an useless local variable ends up in an avoidable pair of `stloc` and `ldloc` CIL opcodes). The resulting code is no longer in valid GIMPLE form; this is fine because the code stays in this form only within the pass. This is conceptually (perhaps not only conceptually) similar to what done by the `out-of-ssa` pass; `out-of-ssa` may even be more powerful in doing this, since it operates in SSA form. It may be interesting to move `simpcil` pass before `out-of-ssa` and to avoid any variable elimination in `cil`. To be evaluated.

Here is an overview of how `cil` pass handles some of GIMPLE constructs. Many of them are omitted, for which the emission is straightforward.

- GIMPLE functions are emitted as CIL static methods of `<Module>`.

- Local-scope `VAR_DECL` nodes are emitted as CIL locals, global-scope `VAR_DECL` nodes as static fields of `<Module>`.

- `INTEGER_TYPE`s and `REAL_TYPE`s are translated into their obvious equivalent CIL scalar types. `BOOLEAN_TYPE`s are translated as CIL `int8`. `POINTER_TYPE`s are translated as CIL `native int`.

- Data structures of type `RECORD_TYPE`, `UNION_TYPE`, `ARRAY_TYPE` and `ENUMERAL_TYPE` are emitted as valuetypes with explicit layout. Remark that GIMPLE `ARRAY_TYPE` nodes cannot be emitted as CIL arrays (which are managed arrays, a specific kind of objects). Explicit layout is necessary because layout of structures and unions is already done when code is in GIMPLE form; CIL declarations have to match the size of such data structures.

- Expressions with `INDIRECT_REF` and `ARRAY_REF` nodes are emitted as indirect memory accesses. Remark that CIL-simplified GIMPLE only allows `ARRAY_REF` nodes with zero offset.

- Expressions with `COMPONENT_REF` nodes are emitted as field accesses.

4 Status

The back-end already supports most of C99 standard and it is validated every night with GCC testsuite.

The only C99 features currently unsupported (or partially supported) are:

- Complex types. The compiler itself handles them, but the related library functions are missing.
- Variable-size structures (structures containing variable-size arrays).
- `setjmp` and `longjmp`. They are not trivial to support in CLI; nevertheless, their semantics can be replicated by using protected blocks and exceptions. It is just not done yet.

Most of the testsuite failures are related to features that are not really part of C99 standard. Even if they are GCC-specific extensions to the language, such features are typically supported by a GCC target. The preferred choice for the CLI back-end is to support them as well; however, some of them do not really match the abstraction level of CLI virtual machine. Examples of GCC extensions unsupported in CLI back-end are:

- Inlined assembly (`__asm__` keyword). In the case of CLI machine, inline assembly consists in CIL bytecode. In order to support this feature, a format should be defined that specifies the location of the source arguments and the overall effect on the evaluation stack of the CLI machine of the assembly instructions.
- Section attributes. They do not really make sense for a CLI back-end, since CLI object files are very different from native objects and common section conventions do not apply.
- Attribute `packed`. In most of cases specifying such an attribute works, but it is not very well tested and it is known to break in the presence of bitfields.
- Nested functions. In general they are supported, except when the address of a nested function is taken.

5 Acknowledgements

CLI back-end project is funded by STMicroelectronics. It is conceived, designed and developed in STMicroelectronics Manno lab (Switzerland) by Andrea Bona and the authors of this paper.

The authors would like to thank the FSF and all GCC contributors for all the work done throughout GCC glorious history. They made this present work possible!

References

[1] DotGNU Portable.NET project. `http://dotgnu.org/pnet.html`.

[2] European Network of Excellence on High-Performance Embedded Architecture and Compilation. `http://www.hipeac.net`.

[3] The Mono Project. `http://www.mono-project.com`.

[4] ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *ECMA 372: C++/CLI Language Specification*, 1st edition, December 2005.

[5] ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *ECMA 335: Common Language Infrastructure (CLI) Partitions I to IV*, 4th edition, June 2006.

[6] Free Software Foundation. The free software definition. `http://www.gnu.org/philosophy/free-sw.html`.

[7] The GNU compiler for the Java programming language. `http://gcc.gnu.org/java/`.

[8] David R. Hanson. Lcc.NET: Targeting the .NET Common Intermediate Language from Standard C. *Software: Practice and Experience*, 34(3):265–286, 2003.

[9] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 23271:2006 Common Language Infrastructure (CLI) Partitions I to VI*, 2nd edition, 2006.