A Stack-Based Internal Representation for GCC

Gabriele Svelto[1], Andrea Ornstein[1], and Erven Rohou[2⋆]

[1] STMicroelectronics, Via Tolomeo 1, 20010 Cornerado, Italy
[2] INRIA, Campus de Beaulieu, 35042 Rennes CEDEX, France

Abstract. Complex embedded systems have always been heterogeneous, and it is unlikely that this situation will change any time soon. Still, the huge non-recurring engineering cost of silicon products tends to make more parts of embedded systems programmable. Our research proposes to address this complexity through processor virtualization. We decided to rely on the CLI format, and we developed a GCC back-end for it. Even though we were able to generate reasonable code, we noticed that we were lacking some important optimizations that exploit the evaluation stack of the virtual machine. Since GCC internals do not provide any support for stack-based instruction set, we introduced our own.
We review the limitations of our previous prototype, and we present the data structures of our internal representation, as well as its API. We also describe a number of optimizations that this representation enabled. To exemplify its convenience, we report the code size improvements we obtained with little effort.

1 Introduction

Complex embedded systems must provide a wide range of dedicated and demanding functionalities, such as communication, multimedia and user interface. Given their tight area and power constraints, no programmable homogeneous architectures can provide those functions. Rather, they typically consist of a host microcontroller running the system software, some heterogeneous processors, such as DSP or VLIW, and dedicated hardware.

Instruction set virtualization is a way to address this heterogeneity. The most widespread technologies are Java [9] and the *Common Language Infrastructure* (CLI). CLI is a rich virtualization environment for the execution of applications written in multiple languages (CLI is better known as the core of Microsoft's .NET technology). It is both an ECMA [4] and ISO [7] standard. Among other features, it includes an effective abstraction of instruction sets in the form of a processor-independent stack-based bytecode format.

In previous work [1, 3], we have shown that CLI is an appropriate program representation for deployment of embedded applications, in terms of both code size and performance. Since most embedded applications are written in the C language, and we were unable to find any adequate C compiler for CLI, we originally developed a port of GCC. This work has been presented in [2].

While this compiler produced reasonable code, we noticed that we were unable to take advantage of the evaluation stack, simply because GCC internals

⋆ Most of this work was completed while the author was at STMicroelectronics.

cannot represent it. The contribution of this paper is to present the lightweight stack-based intermediate representation (IR) that we added to GCC. We developed it to better manipulate CLI code, but it could be adjusted for any other representation that makes use of an evaluation stack. Our developments are publicly available in the usual GCC source code repository.

Section 2 explains in more details the limitations of our previous port of GCC and the motivations for our extension. Sections 3 and 4 respectively describe our new internal representation and the new optimization passes we wrote. We then evaluate its effectiveness in Section 5. Finally, we conclude in Section 6.

2 Motivation

The standard GCC compilation flow is the following: the front-end parses the input source code and turns it into an internal tree-based representation called GENERIC [10, 12]. This code is then lowered into the GIMPLE representation to be manipulated by the high-level optimizers. It is then turned back into GENERIC code before being translated into RTL, the register transfer language used by the back-end for low-level optimization.

RTL, unfortunately, is not a good starting point for emitting CLI bytecode: it is not type-rich, and it assumes that the underlying machine is register-based. The most important phases dealing with RTL are designed around this assumption. Previous attempts to coerce RTL into CLI code [11] reported fundamental problems caused by the mismatch between the abstraction levels.

The first version of our CLI back-end [2] was based on the existing GCC infrastructure, with the exception that we bypassed the RTL passes. We processed the final GENERIC/GIMPLE code in order to emit CLI instructions one by one. Complex nodes were expanded into a sequence of simpler operations, then a single depth-first descent was made on the trees, directly printing CLI code into an assembly file. This technique worked but still it had some drawbacks.

- The simplification pass tried to twist GENERIC so that we could emit assembly directly from it. This was an awkward and fragile process because it relied on implicit assumptions between the simplifier and the emitter regarding how GENERIC nodes were handled.
- The code quality depended heavily on the source trees. No context was available when recursively expanding one node at a time. Shallow trees (like those produced with optimizations off) produced CLI code that did not use the stack at all, forcing us to create many temporaries and bloating the code. Some GENERIC-to-GIMPLE simplifications also caused this behavior.
- We missed the optimizations done on the RTL representation and we could not do CLI specific optimizations as our IR was not really CLI.

These problems prompted us to design a new simple and specific IR which improved code generation and made it more robust and simpler to extend. We refer to it simply as *CLI instructions* or *CLI statements*.

```
struct cil_stmt_d GTY (()) {
  /* Opcode of this statement.  */
  ENUM_BITFIELD (cil_opcode) opcode : 8;
  /* ... */
};
typedef struct cil_stmt_d *cil_stmt;
```

Fig. 1. Representation of a CLI Statement

3 Description of the New Representation

For each funtion, the virtual machine executing the bytecode has a representation of local variables and arguments, and an evaluation stack which is used for storing partial results [4, 7]. Almost all CLI instructions directly operate on the stack, popping their arguments from it and storing back their result. No operation can be done directly on local variables and arguments.

On top of this, the CLI bytecode has rich type information. Pointers retain the type of the object they point to, stack slots retain the type of the object which was stored in them, aggregate types know their fields, etc.

3.1 Representation of CLI instructions

We loosely modeled the CLI instructions against GIMPLE tuples.[3] There are substantial differences, but most of the philosophy is retained. We kept an eye on the `gimple-tuples` branch during its development, because it has become the middle-end representation, and we wanted our design to be immediately familiar to GCC developers. Note that we only used a subset of the instruction repertoire defined by CLI [4, 7]: we focused on *unmanaged* code, specifically on the C language, whereas many CLI instructions are designed for an object-oriented *managed* environment (exceptions, garbage collection, ...)

A basic CLI instruction is made of a single small structure holding the instruction type, an extra type-dependent union field, plus a few extra fields for housekeeping (see Figure 1). The union field is either directly used by simple instructions or it points to another external structure holding more information for complex ones. Instructions working on local variables or arguments use this field for storing a tree representing the `VAR_DECL` or `ARG_DECL` involved; a load- or store-field instruction uses this field to point to a `FIELD_DECL`. This is somewhat similar to GIMPLE tuples statements. Complex CLI statements like function calls or switches use this field for holding a payload of additional information.

Our CLI instructions and their operands use GCC garbage collection facilities like other data structures in GCC front- and middle-end.

[3] Note that the CLI bytecode is actually named CIL (*Common Intermediate Language*). This is the reason why we use the `cil` prefix in our naming convention. However, for the sake of simplicity we only use CLI in the text of this paper.

```
typedef struct { /* ... */ } cil_stmt_iterator;
static inline cil_stmt_iterator csi_start (cil_seq);
static inline cil_stmt_iterator csi_start_bb (basic_block);
static inline cil_stmt_iterator csi_last (cil_seq);
static inline cil_stmt_iterator csi_last_bb (basic_block);
static inline bool csi_end_p (cil_stmt_iterator);
```

Fig. 2. Iterators on CLI Statements

3.2 CLI container structures

CLI instructions are held into sequences of instructions. A sequence node is allocated for each CLI instruction so that the sequence can be easily manipulated. The functions provided for the CLI sequences have names and behavior very similar to their GIMPLE siblings. Sequences are cheap to allocate and manipulate, and they are cached to minimize allocation costs and memory footprint.

GCC provides hooks for additional representations. CLI sequences should be attached to basic blocks like trees or RTL sequences. However, at this time, our port is not integrated in the main development branch, and we did not want to pollute GCC global structures with target-specific data. We used an external hash-table to map CLI sequences to the basic blocks. This allows us to keep our representation almost completely isolated inside our target, and it facilitates the merging process from the main branch.

Generic control-flow information and helper functions can be used transparently so that CLI-specific optimization passes look very similar to GIMPLE passes. However, once the CLI code is generated, only minimal changes to the control flow graph are allowed, as described in Section 4.1.

3.3 CLI instructions manipulation

A rich set of helper function is provided for creating and manipulating CLI instructions. Building new CLI instructions is cheap. While it is possible to directly alter CLI instructions, the preferred method of altering significantly an instruction is to simply replace it. The internal structure of CLI instructions is shielded from the use by a set of accessors. Reuse of GENERIC/GIMPLE operands inside the instructions makes handling or building complex statements easy. Always faithful to the GIMPLE tuples philosophy we decided that manipulation of CLI sequences and instructions would be done through a set of iterators with very similar names and functionality (cf. Figure 2).

3.4 Stack representation

As mentioned before, the CLI bytecode uses a typed evaluation stack for storing partial results. Knowledge about its depth, what variables have been pushed on it, what are the types of its slots in a specific point of the instruction stream are important to implement CLI specific optimizations.

```
struct cil_stack_d {
  /* ... */
};
typedef struct cil_stack_d *cil_stack;
extern void cil_stack_after_stmt (cil_stack, cil_stmt);
extern size_t cil_stack_depth (const_cil_stack);
extern cil_type_t cil_stack_top (const_cil_stack);
```

Fig. 3. Representation and Manipulation of the Stack

Calculating the contents of the evaluation stack over a whole function is a fairly trivial task because of the very nature of CLI code. By design, a single pass over the basic blocks is sufficient for obtaining the full stack information (see § III, 1.7.5 of [4]).

However, an optimizer might want to significantly alter the code while it is working on it. Precalculating all the stack information might not be the best way of dealing with it. Moreover, calculating stack information requires valid CLI code, but the optimizer might want to temporarily leave part of the code in an invalid state, yet still have stack information for its next steps.

To solve these problems we decided to create a simple set of functions which allow an optimization pass to create stacks from basic blocks and propagate them to their successors without calculating directly the contents of the stack. The stack information is updated simply by submitting to it the statements which need to be simulated. This allows for very cheap stack analysis both in terms of computational and spatial resources. See Figure 3 for an excerpt of the stack manipulation functions.

At this point, the information we provide on the stack contents is still partial, even though it was more than enough to implement the optimizations we planned. A full data flow analysis (DFA) would provide accurate information, possibly integrating the stack facilities with GCC DFA infrastructure.

3.5 CLI types and interaction with the GCC type systems

The GCC type system is very complex because it is designed to model types coming from a plethora of input languages. This causes some problems when emitting those types in CLI form. The CLI type system is fairly rich and complete, however its constructs are not enough to entirely cover GCC types.

CLI offers a few basic type categories on which we mapped all other types. A fairly wide range of scalar types are offered, including 8-, 16-, 32- and 64-bits integer (signed and unsigned), pointer-sized integers (also known as "native" integers), single and double precision floating-point types, and typed pointers. Structured types — *value type* in CLI speak — can be loosely compared to C structs but with much more control and different semantics; we used them for modeling all kinds of non-scalar types.

Value types are composed of an arbitrary number of fields and the offsets of those fields can be specified explicitly. Regular structs are mapped directly on

value types; unions are mapped by setting the offsets of the different fields so that they overlap. Complex numbers and vectors are also represented as value types.

Fixed-size arrays and C 0-terminated strings are mapped to value types as well because CLI does not offer an equivalent of a fixed size *unmanaged* array. We could not use CLI arrays because they are managed garbage-collected objects with range-checking. No pointer arithmetic is possible on them. C arrays of unknown size at compile time are mapped to pointers.

We also modeled C99 [6] and GNU-C extended complex types as value types made of two fields of the appropriate basic type. GCC already lowers operations on complex types as operations on their single parts, CLI supports manipulation of value types within variables or on the stack directly, so the expansion of those accesses is extremely simple. This kind of implementation has the upside of preserving the semantics of complex types very clearly in the resulting code.

CLI assemblies (the "executable object" format used by CLI) also requires that every type must be named. We provide a mechanism for automatically generating names for nameless types including the types that we artificially create.

Functions also require some additional processing. C specifies function types (function pointers) but does use them only in the language. Once an object file has been created, functions are referenced only by their names as symbol in the executable object. CLI, on the other hand, requires that functions be completely specified: the name of a function is its complete signature (including the types of its arguments). This causes problems when supporting C89 with K&R style function declarations.

4 CLI-Specific Compilation Flow and Passes

As mentioned in Section 2, RTL and CLI do not match well. The compilation pipeline we are using thus skips all the RTL passes as well as the passes converting GIMPLE/GENERIC into RTL, and adds passes which convert GIMPLE/-GENERIC into our specialized stack-based IR, optimize it and finally emits CLI assembly code from it. The changes we made did not impact the usual compilation pipeline and the changes done into `passes.c` are enabled only if the CLI target is selected, leaving the other passes in place if a conventional target is used. As soon as a flexible pass manager is available, we will be able to re-implement these changes without touching `passes.c`. The rest of this section describes in detail the various passes of the CLI compilation flow.

4.1 Basic blocks reordering and control flow graph interaction

The evaluation stack plays an important role when it comes to control flow. Every basic block in a piece of CLI code can be thought as having a stack representation associated with it, holding the depth and types of the stack at the entry point. Every branch to the basic block must also have a stack with

the same number of slots filled and compatible types for those slots. However, there is an additional constraint, the CLI code must be verifiable (i.e. checked for consistency) in a single linear pass (see § III, 1.7.5 of [4]). This forces basic blocks which are reached only by backward branches to have an initial empty stack.

These constraints make manipulating the control flow complex. Moving basic blocks or redirecting branches causes changes which propagate over the control flow graph and which would be very complex to deal with.

For the sake of simplicity, we introduced an initial pass which reorders basic blocks to minimize the number of jumps. We then "freeze" the CFG and avoid further changes which might change the order of the basic blocks (or the branch directions relative to them).

4.2 Switch expressions break up

The CLI `switch` instruction implements the multi-way control-flow statement present in many languages. However, it cannot be used directly for encoding any C `switch` statement. The set of "cases" of the CLI `switch` is a range of adjacent values starting at zero. For each value in that range, a branch destination is stored. Using this construct to implement sparse switches thus would waste a lot of space in filling the "empty" slots with branches to the default label. On top of this, the `switch` instruction cannot specify a range larger than 8192. Larger switches — even if dense — must be broken up into multiple `switch` instructions. We added a pass before GCC builds the CFG that analyzes the `switch` statements and breaks them into smaller `switch` and conditional expressions in order to work around those constraints.

4.3 GIMPLE/GENERIC to CLI expansion

We lower GIMPLE/GENERIC code to CLI code by descending the original trees and expanding one node at a time. Many GIMPLE statements can be expanded into a single CLI instruction making the code generation fairly straightforward with each node being expanded and implicitly leaving its result on the stack; however some statements require significantly more effort. Unordered floating-point comparisons, bit-field extractions and insertions, rotations and some boolean expressions are turned into fairly large CLI sequences. Before the new representation was available, those statements were rewritten in GIMPLE, generating a large number of temporaries; now we are able to store the temporaries on the stack fairly often.

We also deal with many built-in functions in this phase. Some of GCC's built-ins are turned into CLI built-ins provided by the support library which ships with our back-end. Any JIT compiler aware of the semantics of those built-ins can directly take advantage of them. Other JITs will have to rely on the runtime library. Some built-ins which can be mapped efficiently to CLI instructions and are expanded in this phase. For example `__builtin_memcpy` is turned into a `cpblk` instruction.

This phase was designed to improve on our previous work and generate better code. However, the quality of the resulting CLI code still depends on the input code. Deep GENERIC trees result in excellent and compact code making heavy use of the stack. Shallow trees and GIMPLE temporaries result in fat code and large metadata sections needed to encode CLI local variables. Instead of trying to fix up these inputs up front, we introduced some later optimizations passes to clean up these code sequences. This has the side benefit of allowing us to make this critical phase emit "naive" code, thus keeping it simple and robust.

4.4 Missing prototypes fix-up

This is not an optimization phase, but it is necessary for fixing up functions which do not have a prototype. When such a function is called, the types that it accepts are treated in the same way as the arguments of a variable argument function (i.e., `char` and `short` are promoted to `int`, etc.) However, the function's actual arguments can be of unpromoted types. The function signature in CLI uses the promoted types so that the linker can find it correctly and this pass creates artificial local variables with the correct argument types, converts the passed arguments and stores them in those variables. Those locals are then used instead of the arguments. The fix happens only if the promoted argument type is different from the actual type, other arguments are left alone.

4.5 Redundant temporaries removal and stack promotion

The middle-end tends to generate many temporaries. The normal compilation flow of GCC removes many of them during register allocation. In our case, however, there is no register allocation. This new phase scans the code and locally tries to identify temporary variables and to promote them to the stack.

Ideally, a full data flow analysis is needed to extend this phase across basic blocks. Yet, we implemented this phase as a sort of improved peephole optimizer, and we obtained very good results with very little effort. The pass works as follows:

- We scan for variables whose address has been taken and we tag them. They remain untouched.
- We then look for `stloc` instructions (store to local variable). For each of them, the depth of the stack is recorded and a matching `ldloc` (load local variable) is looked for. If the stack has not been altered below the depth of the `stloc` between the two instructions and if the depth is the same at both instructions, the `stloc` is replaced by a `dup stloc` sequence and the corresponding `ldloc` is removed. If the value on the stack is an integer larger than the temporary variable type, a conversion is put before the `dup` in order to simulate the truncation caused by writing to the variable.
- The code is scanned a second time and other simple copies which cannot be promoted on the stack are identified. This involves finding `ldloc`/`stloc` or `ldarg`/`stloc` pairs and replacing the load from the target variable with a

load from the source variable if it has not been modified in between. This step can be seen as a very simple copy propagation mechanism and works across basic-blocks.
- Finally, dead stores left by the previous steps are removed.

Consider the GIMPLE code of Figure 4. Figure 5 illustrates the transformations of this pass: (a) is the initial IR generated by the middle-end. In (b), the variables t1 and t2 are clearly redundant and they are promoted to the stack. In (c), the variable c is copied into t3. Finally, in (d), the dead stores are removed.

Note that this transformation is already profitable after step (b) because the `dup` instruction is encoded on one byte, while the `ldloc` needs two or four bytes (except for the special case of variables numbered 0 to 3, where one byte is enough and the transformation is neutral).

```
t1 = a + b;
t2 = p->foo
t3 = c
t4 = (t2 - t1) + t3
e = t4
```

Fig. 4. Original GIMPLE Code

(a)		(b)		(c)		(d)	
ldloc	a	ldloc	a	ldloc	a		
ldloc	b	ldloc	b	ldloc	b		
add		add		add			
stloc	t1	dup		*dup*		ldloc	a
ldloc	p	stloc	t1	*stloc*	*t1*	ldloc	b
ldfld	p->foo	ldloc	p	ldloc	p	add	
stloc	t2	ldfld	p->foo	ldfld	p->foo	ldloc	p
ldloc	c	dup		*dup*		ldfld	p->foo
stloc	t3	stloc	t2	*stloc*	*t2*	sub	
ldloc	*t1*	ldloc	c	*ldloc*	*c*	ldloc	c
ldloc	*t2*	stloc	t3	*stloc*	*t3*	add	
sub		sub		sub		stloc	d
ldloc	t3	*ldloc*	*t3*	ldloc	c		
add		add		add			
stloc	e	stloc	d	stloc	d		

Fig. 5. Optimization Steps

4.6 Redundant conversions removal

GIMPLE conversion nodes (`CONVERT_EXPR` and such) are often expanded into actual conversions on the stack because we process one node at a time and we do not have enough visibility to decide when a conversion is redundant.

We first remove the conversions which behave as no-ops simply looking at the types of the stack slots. Then, we also try to remove conversions made useless by the particular use of their result. For example, when storing a stack slot to

a local variable or memory location, holding an integer smaller than 32-bits, an implicit truncation is done. If a conversion from a 32-bit integer to a 16- or 8-bit integer precedes a store to an 8-bit integral local variable it is removed.

4.7 Peephole optimizations

This pass is used for cleaning up simple instruction patterns generated by the previous passes and are fairly simple to deal with. Currently this phase mostly removes naive instruction sequences emitted when dealing with functions with a variable number of arguments. These instruction sequences usually involve putting the address of a local variable or argument on the stack and then writing or reading indirectly into the variable using this address. These patterns can be easily removed and replaced by regular loads and stores.

4.8 Branch condition replacement

In this pass, the conditional branches at the end of each basic block are analyzed and potentially changed in the aim of reducing the code size. If one of their edges goes to the next basic block, the pass checks if the instruction can be emitted in a single compare-and-branch instruction with the edge going to the basic-block becoming a fall-through. If this does not happen this pass will try to invert the branch condition and the outgoing edges to make the fall-through possible.

4.9 CLI assembly emission

The last pass emits the CLI code. It is fairly straightforward as the IR maps one-to-one with the CLI code. This phase, however, contains a few extra functionalities which are not usually required by an assembler emission phase. After all the functions have been emitted, this pass proceeds by emitting the type declarations, as well as the global variables, static variables (including function-static ones which cannot be declared inside a CLI function and thus must be "promoted" as plain global variables) and the string constants.

5 Examples and Experiments

This paper is not about quantitative achievements. Rather, it is about presenting the new IR we added to GCC, its design and its purpose. Still, as an illustration of its convenience, we quickly report the improvements that we have been able to achieve with a minimal effort. In the interest of space, we do not report in this paper the code of the optimizations themselves. However, all the above mentioned transformations are fully implemented and publicly available in the branch `st/cli` of the GCC Subversion repository.

The code transformations we implemented so far mostly target the code size. Performance is less impacted because a JIT will eventually transform the CLI to

native code, replacing stack elements with machine registers. Classical optimizations applied by the just-in-time compiler will remove some of the inefficiencies of the bytecode[4].

We used benchmarks from the MiBench [5] and MediaBench [8] suites, as well as a few proprietary multimedia benchmarks. All benchmarks were compiled at optimization levels `-O2` and `-Os` with and without our new IR. Figure 6 shows that the modest effort made at implementing the above mentioned optimizations immediately pays off for most benchmarks. The small degradations actually are a bug fix: the previous implementation was too aggressive at removing conversions, and in some corner cases it generated incorrect code.

While the code size may be of relative importance to the general purpose computing, it does matter for embedded systems because the non volatile memory is a significant part of the total cost, and therefore it must be carefully sized. Anyhow, our proposed IR provides a means to manipulate a stack-based instruction set. As such it can also be used to implement optimizations targeted at performance.

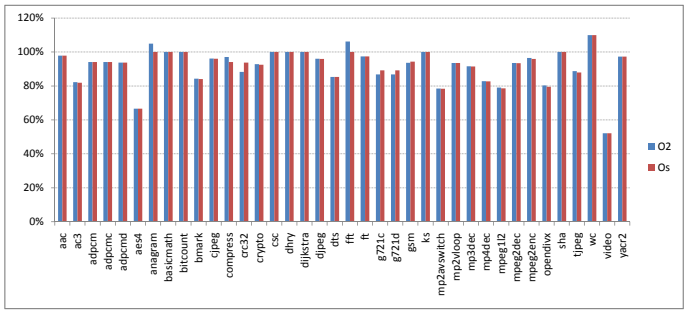

Fig. 6. Relative Code Size

6 Conclusion

In the course of our research, we are interested in applying virtualization technologies to ease the programming of embedded systems. In particular, we considered the CLI format. We have been developing an initial port of the GCC compiler for this purpose. While we were able to generate reasonable code, we noticed that we were unable to take advantage of the evaluation stack, simply because the GCC internals cannot represent it.

This paper presents the new lightweight stack-based intermediate representation that we added to GCC for this purpose and made publicly available in

[4] An *interpreter* would be faster because of the reduced number of instructions. The JIT compilation time can also benefit from these improvements.

the GCC source repository. We describe the data structures representing CLI instructions and the helper functions, as well as the stack representation. As an illustration, we also present a number of simple optimizations that we wrote using our new IR, and we report the code size improvements we achieved.

We strongly believe in the benefits that virtualization can bring to embedded systems. Being able to generate optimized CLI code from the C language is a mandatory step towards our goals. Our contribution clearly shows the need for exploiting the evaluation stack, and the benefits one can draw from our stack-based intermediate representation.

References

1. Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea Ornstein, and Erven Rohou. An experimental environment validating the suitability of CLI as an effective deployment format for embedded systems. In *Conference on HiPEAC*, pages 130–144, Göteborg, Sweden, January 2008. Springer.
2. Roberto Costa, Andrea C. Ornstein, and Erven Rohou. CLI back-end in GCC. In *GCC Developers' Summit*, pages 111–116, Ottawa, Canada, July 2007.
3. Roberto Costa and Erven Rohou. Comparing the size of .NET applications with native code. In *3rd Intl Conference on Hardware/software codesign and system synthesis*, pages 99–104, Jersey City, NJ, USA, September 2005. ACM.
4. ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *Common Language Infrastructure (CLI) Partitions I to IV*, 4th edition, June 2006.
5. Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4th Annual Workshop on Workload Characterization*, pages 3–14, Austin, TX, USA, December 2001. IEEE.
6. International Organization for Standardization and International Electrotechnical Commission. *International Standard ISO/IEC 9899:TC2 – Programming languages – C*, 1999.
7. International Organization for Standardization and International Electrotechnical Commission. *International Standard ISO/IEC 23271:2006 – Common Language Infrastructure (CLI), Partitions I to VI*, 2nd edition, 2006.
8. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–335, Research Triangle Park, NC, USA, December 1997.
9. Tim Lindholm and Franck Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.
10. Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proceedings of the GCC Developers Summit*, pages 171–179, Ottawa, Canada, May 2003.
11. Jeremy Singer. GCC .NET - a Feasibility Study. In *1st International Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Distributed and WEB Computing*, volume 1, pages 55–62, Plzen, Czech Republic, February 2003.
12. Richard M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation.