Selecting Compiler Platform for Lissom

Miloslav Trmač

April 11, 2008

1 Overview

Three main C compiler candidates were chosen: `gcc`[1], LCC[2], and LLVM. This document summarizes my study of their design, their implementation and their capabilities for the purpose of implementing a Lissom compiler back-end generator for one of the compilers.

2 LCC

`gcc` and LLVM require more detailed description, but LCC is quite simple: A hand-written lexer and parser parse each C function into a DAG. Local transformations (local expression simplification, constant folding and local common subexpression elimination) are performed on the DAG. A bottom-up dynamic programming tree pattern matcher generated from a machine description is used to find instructions that implement each DAG; the machine descriptions often directly map a single expression tree node to a sequence of two or three consecutive instructions.

The result of the bottom-up tree automaton is a linear sequence of instructions. Register allocation[3] works one DAG at a time, allocating physical registers in the order physical registers appear in the instruction sequence. If a register needs to be spilled (because no free registers are available or because an instruction sequence references a physical register), a register that won't be used for the longest time in the future is chosen.

LCC does not implement instruction scheduling, any global optimization, nor other "advanced" techniques.

The following sections describe only `gcc` and LLVM.

3 Front-end

Both `gcc` and LLVM use the `gcc` front-end, with a hand-written lexer and a hand-written recursive descent parser. The front-end supports several languages, often in multiple variants (including C90, C99, "GNU C", C++ and ADA). The front-end also supports, at least in the case of C a very large set of extensions to the standard language. Some, like local labels, are quite trivial and transparent to back-ends, others, like fixed-point and decimal floating-point types, affect everything up to back-end.

gcc obviously supports all of the features of its own front-end. LLVM uses a patched version of gcc as the front-end, so the gcc version used currently lags behind the newest gcc version, and correspondingly the supported features of the LLVM front-end lag behind the features of gcc. Except for this time lag, LLVM supports most of the C extensions (notably it supports the complex gcc-style asm statements). The C extensions that are not completely supported (__builtin_apply, nested functions, attribute(malloc), attribute(no_instrument_function) are very rarely used. The LLVM code representation can support separate address spaces and arbitrary-width integers, and patches that add support for these features to the front-end are supposed to exist. Note that the gcc front-end refers to target-specific data, so any LLVM target must also be supported, at least somewhat, as a gcc target.

To summarize, LLVM has no significant disadvantages and at least a potential for significant advantages in front-end capabilities for embedded software development. This is partially offset by the necessity to generate two separate target-specific back-ends, perhaps slowing down the edit-compile-debug cycle a lot.

4 Middle-end

Both gcc and LLVM implement the most important global optimization algorithms like partial redundancy elimination, conditional constant propagation and dead code elimination.

In general, gcc has better implementation of most global optimizations; in particular, it has a much better loop optimizers (which support modulo scheduling, various loop nest transformations and automatic vectorization). LLVM was designed from the beginning for interprocedural and link-time optimization, and supports a few more interprocedural optimizations than gcc does; on the other hand, its link-time optimization support is documented not to work 'on most platforms "out-of-the-box"'.

Neither gcc nor LLVM has a very significant advantage over the other compiler in this area: the advantage of LLVM over gcc in interprocedural optimizations is not as large as one would expect after only reading documentation, and most loop nest optimizations can be performed by hand if necessary. Manually converting code to use vector instructions may be difficult (it usually requires specific knowledge of the target vector instructions and their corresponding gcc built-ins), however, which gives gcc a small advantage.

5 Back-end—Representation

The differences in code representation in compiler back-ends is considerable. In RTL, the gcc representation, a semantic description of instruction operation is primary; instructions which implement the semantics are later selected and attached to the semantics description. RTL value types ("machine modes") are quite rigid: although bytes with more than 9 bits are supported, the size of other types must be specific multiples of the byte size. For example, supported integer widths are only 1, 2, 4, 8, 16, 32 complete bytes. On the other hand, RTL supports a significantly wider range of operators than the operators that are necessary for C implementation, e.g. the minimum and maximum operators.

LLVM treads machine instructions, once generated, as opaque, described only by a list of input and output registers (the target-specific code can access target-specific attributes and refer to instructions by name, but the generic code does not). Notably the representation uses the SSA form even after conversion to machine instructions, until register allocation is finished. The LLVM type system is much more flexible for integers—integers of literally any width are supported—but, like `gcc`, it supports only a small fixed set of floating-point format sizes.

The `gcc` representation is quite complex, handling e.g. debug information and exception-handling information directly. The LLVM representation is formally much smaller and simpler, representing some of the "auxiliary" data as calls to built-in functions. The auxiliary data thus does not pollute the main language, and optimization passes can simply treat the built-ins as ordinary function calls (assuming unpredictable side effects on global data, as usual), which makes at least some of them much simpler; unfortunately, this also means that if code could be significantly transformed, the original debug information prevents the transformation (in particular dead code elimination is completely ineffective because it does not recognize the debug information built-ins). Therefore, LLVM only emits debug information with `-O0`. It could be argued that the debug information generated by `gcc` when optimizations are enabled is not that useful; on the other hand, any debug information is better than nothing when a customer provides a crash dump.

The flexibility of LLVM integer types is a significant advantage that `gcc` probably will not be able to replicate in near future.

6 Back-end—Code Generation

At a first glance, LLVM with its tree pattern matching is clearly superior to `gcc`'s pattern expansion and instruction combination[4]-based code generator. The true situation is quite different, though.

`gcc` code generation is based on a machine description file, which describes all relevant instructions, specifying for each instruction the equivalent RTL expressions, code to generate assembler output for the instruction and operand constraints that must be satisfied to make the instruction valid. Code generation starts by generating quite naive code, replacing each tree node by a RTL pattern with a corresponding well-known name (e.g. `addsi3` for addition of two 4-byte integers). Some well-known names are mandatory, some are optional; the translator to RTL and later optimization passes implement alternative code generation strategies that depend on existence of specific named instructions.

After some optimization passes, RTL expressions are matched against instructions, using the first instruction in the file that matches the RTL expression and whose constraints are satisfied by the RTL expression. A pass specific to RTL-style compilers is *instruction combiner*, a generalized peephole optimizer, which combines RTL expressions of two or three "related" instructions, simplifies the result, and searches for an instruction matching the RTL (which can be used to replace the original two or three instructions).

The LLVM does use a pattern matching code generator, but it doesn't use any of the standard bottom-up dynamic-programming algorithms: instead, instructions matching a specific expression tree are selected top-down, using the first one that is applicable (based on user-specified conditions) out of a list that is ordered by "pattern complexity" and instruction cost. Thus, the generated code can be only locally-optimal at the level of a single instruction—and it actually is not, because each instruction is considered to have the same cost. The LLVM code generator therefore is roughly equivalent to the `gcc` code generator: it starts with a tree describing the expression (RTL in `gcc`, a `SelectionDAG` in LLVM), and uses the first applicable instruction that matches the root of the expression.

Unlike `gcc`, which handles operations not supported by hardware by using alternate code generation strategies in the generic code, LLVM targets must specify a "strategy" for handling each unsupported operations. The most common strategies (using a larger type, using a different type, using a set of smaller types) are implemented in the generic code, other strategies need to be implemented manually.

Overall, `gcc` has a non-trivial, although not extreme, advantage over LLVM in code generation. The long history of `gcc` and its very large number of currently or formerly supported architectures suggest that the generic code in `gcc` supports most reasonable strategies of handling operations that are not supported by the hardware. The `gcc` instruction combiner is perhaps slow and its technology is considered obsolete, but the LLVM attempt at tree pattern matching is certainly not stronger than the `gcc` instruction combiner.

7 Back-end—Register Allocation

The `gcc` register allocation algorithm predates widespread use of graph-coloring register allocators.[5] The machine description contains a set of constraints for each operand of each instruction, which specify the required and suggested register classes for each register operand. The allocator collects these requirements for each virtual register, uses it to select a register class, and attempts to allocate a register for that class; local allocation is performed for each basic block, then global register allocation. If a register needs to be spilled, or the allocated register does not match constraints of some instruction, a "reload" pass attempts to fix it by replacing the instruction or adding load/store/move instructions. Overall, the `gcc` register allocator is similar enough to a "true" graph-coloring register allocator, yet different enough to avoid the patents on graph-coloring register allocation.[6] Several attempts to replace the `gcc` register allocator by a "true" graph-coloring allocator were made, but none of the register allocators both was stable enough and generated no worse code than the current `gcc` register allocator.

LLVM provides three register allocators: s trivial "register allocator" that does not keep any values in registers, a local register allocator, and a global linear-scan[7] register allocator, which represents live ranges as intervals over a linear sequence of instructions. The article originally describing linear-scan register allocation reports that the generated code is up to 10% slower than code generated using a good graph-coloring register allocator; because a graph-coloring register allocator is not available in LLVM, direct comparison is not possible.

Good data for comparing the two approaches is not available; the data that is available suggests that the gcc register allocator is better.

8 Back-end–Scheduling

The gcc scheduler is quite full-featured: it uses a DFA for pipeline hazard detection (or multiple separate DFAs to avoid combinatorial explosion problems). It can represent instructions that can use several alternative execution paths depending on execution unit availability, instruction latencies different from the time necessary to process the pipeline, special-case instruction latencies between specific instruction combinations (to represent short-cuts in the pipeline). A notable, although complex, feature, is support for automatic relations among availability of execution units, which can be used to implement constraints on allocating instructions to VLIW bundles, e.g. in the IA64 back-end.

The LLVM scheduler is weak in comparison: the list of execution units used by an instruction is fixed, the latency between instructions is assumed to be equal to pipeline processing time, hazard detection is performed at compile-time, and there is no support for VLIW scheduling (the IA64 back-end only adds bundle separators between instructions to preserve data dependencies, and leaves bundle formation on the assembler).

The gcc scheduler is clearly superior.

9 Other considerations

gcc is available under the GPL, which requires distributors to make the source code of any derived work available along with binary programs. LLVM is licensed under a variant of the 3-clause BSD license, which does not make such a requirement, but it depends on llvm-gcc, the front-end, which is licensed under GPL. A functional program that generates the gcc target code necessary to run llvm-gcc would therefore have to be provided including the source code; the parts that generate the LLVM back-end, if they can be separated from the gcc back-end generator, might be published without the source code. Tradition and common courtesy suggests making the complete source code widely available anyway—especially because large part of the newly written code would probably consist of improvements to the generic LLVM code, and pushing the changes upstream decreases the effort necessary to maintain it.

gcc is quite old and written in C, so some data structures are quite complex for efficiency, and the code needs to be quite verbose; LLVM is written in C++, and the code is usually smaller and cleaner, especially for very simple optimization passes. A rough comparison of LLVM targets with the corresponding gcc targets shows that the target-specific code is roughly five times as large in gcc than in LLVM—but this comparison does not take into account any difference in supported platforms and functionality (LLVM is not around long enough to have accumulated many target-specific optimizing transformations). On the other hand, reading the LLVM back-ends seems to suggest that they contain quite a lot of boilerplate code or code that could probably be placed in the generic parts of the compiler.

gcc been ported to many more platforms than LLVM, and the total effort to optimize the gcc back-ends has been much larger than the effort to optimize LLVM back-ends; it is likely that both the LLVM language and its data structures would become at least somewhat more complex and less clean if they had to support all features of the current `gcc` back-ends. It is still noticeable that the LLVM back-end interface was originally intended to support completely separate back-end implementations with little or no common code.

10 Conclusion: Suggested Action

LCC is small and simple, and creating a Lissom back-end generator for LCC would clearly take the smallest amount of time, but this would result in a compiler that generates quite inefficient code, inferior to other alternatives. Adding the necessary global optimizer to bring LCC up to par to LLVM or `gcc` is technically possible, but completely impractical.

If one had to write a compiler from scratch, the LLVM language and compiler seems better-designed than `gcc`; but given the existing implementations, `gcc` has more features than LLVM, especially in the generic back-end code. Planning to write a LLVM-based Lissom C compiler back-end needs to include planning to implement some features that are already present in `gcc`, such as a VLIW-aware scheduler.

LLVM has two significant advantages—separate address space support and arbitrary-width integer support. Because `gcc` uses plain C types (e.g. `int`) to represent integer values and RTL types, a reliable implementation of these features in `gcc` would require a lot of effort and testing. Due to the large extend of the changes and the comparative insignificance of the features (in light of the fact that it was apparently not necessary to implement any of the existing `gcc` back-ends), it is not very likely to be accepted upstream, so implementing these changes in `gcc` would probably require perpetual maintenance by Lissom developers.

Thus, the suggested action is to use LLVM if either separate address space support or arbitrary-width integer support is considered essential, and `gcc` otherwise.

References

[1] Richard M. Stallman and the GCC Developer Community: GNU Compiler Collection (GCC) Internals, versions 4.1 and 4.3

[2] Christopher W. Fraser, David R. Hanson: The lcc 4.x Code-Generation Interface, Microsoft technical report MS-TR-2001-64

[3] Christopher W. Fraser, David R. Hanson: Simple Register Spilling in a Regargetable Compiler, Software—Practice and Experience, January 1992, Vol. 22(1), pp. 85–99

[4] Jack W. Davidson, Christopher W. Fraser: Code Selection through Object Code Optimization, ACM Transactions of Programming Languages and Systems, October 1984, Vol. 6(4), pp. 505–526

[5] G. J. Chaitin: Register Allocation and Spilling via Graph Coloring, Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, pp. 98–105

[6] Jan Hubička, personal communication

[7] Massimiliano Poletto and Vivek Sarkar: Linear Scan Register Allocation, ACM Transactions of Programming Languages and Systems, September 1999, Vol 21(5), pp. 895–913