MieruCompiler: Integrated Visualization Tool with “Horizontal Slicing” for Educational Compilers

Katsuhiko Gondow  
Tokyo Institute of Technology  
gondow@cs.titech.ac.jp

Naoki Fukuyasu  
Wakayama University  
fukuyasu@sys.wakayama-u.ac.jp

Yoshitaka Arahori  
Tokyo Institute of Technology  
arahori@sde.cs.titech.ac.jp

ABSTRACT
This paper proposes a novel visualization tool for educational compilers, called MieruCompiler. Educational compilers that generate native assembly code like i386 have many practical and pedagogical advantages, but they also have a disadvantage that the undergraduate students need to acquire a wide range of knowledge on native machine instructions, assembly directives, application binary interface (ABI), and so on. To reduce this learning cost, MieruCompiler provides various visualizations as a rich internet application (RIA) including: (1) highlighting all related slices (called “horizontal slicing” after [13], but not implemented in [13]) among the source code, abstract syntax tree, assembly code, symbol table, stack layout and compiler code, when the user hovers the mouse pointer over a piece of them, (2) displaying tooltips for machine instructions, assembly directives, etc., and (3) visualizing stack layouts which are very likely to be implicit. As a preliminary evaluation, MieruCompiler was used in two universities, which produced promising results.

Categories and Subject Descriptors
K.3 [Computers and Information Science Education]: Computer Science Education.

General Terms
Design, Languages

Keywords
Compiler course, Code Generation, Visualization, Program Understanding, Rich Internet Application (RIA)

1. INTRODUCTION
For computer science undergraduate students (students for short), compilers are common and important at most universities [1, 2]. In the compiler courses, educational compilers are often used to provide hands-on experience through compiler construction. PL/0 compiler [14] is a popular classical educational compiler, and many other educational compilers have been proposed so far [15–18].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE’10, March 10–13, 2010, Milwaukee, Wisconsin, USA.  
Copyright 2010 ACM 978-1-60558-865-8/10/03 ...$10.00.
2. NATIVE COMPILER APPROACH

2.1 Why native compiler approach is better

We argue that native compilers are more pedagogically effective than non-native compilers because of the following reasons:

- Knowledge and experiences obtained from real assembly code will be widely applicable for many other areas including embedded systems, computer architectures, and operating systems. Also they are extensible in the sense that further learning leads students to more advanced topics like real compilers (eg., GCC) and techniques on JIT compilers or optimizers.

- Real assembly code is complex as a result of satisfying real-world constraints and requirements, so learning real assembly code is simply interesting and challenging for students; it is just a kind of happy hacking.

- In some senses, virtual machines are *imitations*, which demotivates students to learn them. For example, most native machines are register-based, while most virtual machines are stack-based. There are many differences and gaps in design decisions on real CPU instruction sets and virtual ones. Furthermore, real CPU instruction sets cannot be modified, while most virtual ones can be easily modified. A mission of real-world compiler is to generate low-level code using a *fixed* instruction set, but this does not often hold true for non-native compilers.

- By comparing the two versions of native assembly code: one generated by *real-world* compilers (eg., GCC) and the other generated by the *educational* compiler, students practically know what code should be generated and what alternatives there are. Even if the technique mentioned in [12] is used, it is still quite difficult for students to understand the internal data structures of real-world compilers. On the other hand, it is not so difficult for students to understand the assembly code generated by GCC.

For this purpose, it is desirable that a source language of the educational compiler is a strict syntactic subset of the existing programming language (eg., C), to make it possible to compare different results with the same source program. This is the reason why XC is a subset of C.

- There are many high quality documents (eg., [20, 23]) and tools (eg., GCC and GNU Binutils) for real machines, while not for most virtual machines.

- Library functions (eg., `printf` in the C standard library) can be directly called from the source program, if the educational compiler conforms to the application binary interface (ABI). This is the case for XCC.

This widens the application area written in the source code of the educational compiler, and gives another reality to the students.

---

1 Mieru means “visible” in Japanese.

2 The compiler code in MieruCompiler can be hidden by option for educational purpose.

---

---

---
2.2 Problems in native compiler approach and our solutions

As mentioned in Section 2.1, there are many important advantages in native educational compilers. Unfortunately, however, the most difficult problem in the native compiler approach is that the learning cost is greatly increased, since students need to acquire a wide range of knowledge on native machine instructions, assembly directives, application binary interface (ABI), and so on. Moreover, data structures for AST, symbol table, and type representation are difficult for students to understand and implement.

To solve these problems while preserving the good essence of the native compiler approach, we took the following three ways:

- We carefully defined the source language of our own educational compiler as a strict syntactic subset of the C programming language, which has the following benefits:
  - This saves the time spent in teaching the syntax and semantics of the source language, since the instructors only have to explain the syntactic difference from C.
  - This helps students more deeply understand the C semantics, for example, about “left-to-right evaluation” [22] in && and || operators, pointer arithmetic in + and – operators, the difference between “call by value” and “call by reference”, and the lifetime of automatic variables. This is required to preserve the meaning of the source program being compiled, which is a very basic compiler’s mission.

Note that PL/0 [14], for example, is not a syntactic subset of Pascal, even though PL/0 is a simplified version of Pascal. For example, no type is specified in PL/0 (var i;), while a type is always specified in Pascal (var i: integer). This implies Pascal compilers cannot compile PL/0 programs, while C compilers can always compile any XC program without modification.

- We emphasize and focus on code generation, but not on parsing and optimization.

For example, we would like to put more emphasis on how to translate a while statement into i386 assembly code in Fig. 2 than on parsing and optimization. This is because in our opinion the most important thing is, for example, to teach a while statement can be implemented by combining a conditional branch (je in Fig. 2) and an unconditional branch (jmp in Fig. 2).

Parsing techniques are often emphasized in compiler courses, since LALR parser generator is regarded as a great victory in computer science (eg., in [19]). Unfortunately, LALR parser generator is not the best solution. The grammar and action rules by GNU Bison (an LALR parser generator) are often complicated, since some kludges using “Actions in Mid-Rules”, etc. are required to handle context dependencies [21]. As a result, the parsers in GCC-3.4 and later are changed to hand-written recursive descent one, while the parser in old GCC was implemented by GNU Bison.

In order to avoid the tedious kludges and to maximize the time for the students to concentrate on code generation, we decided to use the strategy of providing to the students the source code of the XCC front-end (scanner, parser and semantic analyzer) and course materials that explain how to access the AST, symbol table, etc. in the XCC front-end. Optimization plays a great important role in compiler systems. For a first course in compiler construction, however, optimization is too complicated for the students to understand and implement. For this reason, our educational compiler XCC only uses a very naive register allocation method where general-purpose registers are used as a part of stack frame. This keeps XCC very compact.

- We provide a visualization tool for better understanding.

As mentioned before, in native compiler approach, the students need to understand native machine instructions, assembly directives, and internal data structures in the educational compiler itself. The cost of learning and understanding them is very high. There is a limit to reduce the cost by enriching the course materials, or by improving the source code using software engineering techniques (eg., modularization and refactoring). Using a debugger (eg., GDB), students can examine the internal data structures, but they often get lost in the AST with hundreds of nodes. Also, technical documents tend to be large (eg., [20] has about 3,000 pages), so students often get lost in those large documents.

Therefore, we provide a visualization tool to solve this problem, which includes:

- (1) Highlighting “horizontal slices”, that is, all related slices among the source code, assembly code, AST, symbol table, stack layout and compiler code.
- (2) Interactive tooltips that explain machine instruction mnemonics, assembly directives, fundamental architecture components like registers, etc.
- (3) Visualization of the AST, symbol table and stack layout.

This visualization tool is implemented as a rich internet application (RIA), which implies that students do not have to install it.

3. XCC AND MieruCompiler DETAILED

3.1 XC

XC is a very compact language while preserving the high expressive power. XC has only 57 grammar rules in GNU Bison, while C has about 200 ones. XC has most features of C including types (eg., int, char, void, function types, pointer types), operators (eg., =, [, ], &&, &), control structures (eg., if-else, while, goto, return), and function (calls) with parameters.

In XC, we eliminated most features that have alternative ways to write. For example, XC does not have the array operator ([]), but as a substitute for it, we can use malloc and pointer arithmetic in XC. For int a[10]; a[3]=5; in C, we can write int *a; a=malloc(40); *(a+3)=5; in XC.

3.2 XCC

XCC is the educational compiler we developed: XCC translates an XC program into native assembly code (i386 and MIPS assembly code for now5). To keep XC simple, XCC does not have intermediate code nor optimizers. The code size of XCC for i386 is about 2,100 lines in C, GNU Bison and GNU Flex. So far, the projects using XCC have been conducted in Japan Advanced Institute of Science and Technology (JAIST) in 2000, in Tokyo Institute of Technology from 2003 through 2009, and in Wakayama University in 2009.

3.3 MieruCompiler

MieruCompiler is our integrated visualization tool with “horizontal slicing.” MieruCompiler statically visualizes the source code (XC), assembly code (i386 and MIPS), AST, compiler code (optional), symbol table and stack layout in XCC.

5Tooltip is a small message window appearing when the user hovers the mouse pointer over an item, without clicking it.

5XCC was originally developed to generate SPARC assembly code, but SPARC is not supported in the current XCC.
3.3.1 Visualization in MieruCompiler

As mentioned in Section 1, MieruCompiler provides a mechanism of “horizontal slicing.” As an example of “horizontal slicing”, in Fig. 3, the corresponding assembly code is highlighted when selecting a while statement. Note that all the other related slices in the AST, compiler code, symbol table and stack layout are also highlighted in MieruCompiler, but they are not shown in Fig. 3 for the lack of space.

Fig. 4 shows an example of tooltips in MieruCompiler, which shows a compact description about i386 popl instruction (its format, semantics and usage example). In MieruCompiler, a tooltip will appear when the user hovers the mouse pointer over machine instructions, assembly directives (eg., .text), register names (eg., %esp) and any form of addressing mode (eg., 4(%ebp) and r/m32) in assembly code and other tooltips. Note that MieruCompiler tooltips can be nested; the first tooltip contains many items which show a second tooltip. Thus the user can see two or more tooltips simultaneously by hovering the mouse pointer over several items successively.

Fig. 5 shows an example of the stack layout visualized in MieruCompiler, which shows that, from the bottom to the top, the stack consists of two arguments (size and data), a return address, an old value of %ebp pointed by %ebp, two local variables (i and j), and other intermediate values unknown at compile-time; and the top stack is pointed by %esp. This graphical information about the stack layout is quite important to help the students to map a local variable or function argument to its corresponding location on the stack relative to the base pointer (%ebp for i386). This is necessary because application binary interface (ABI), including calling convention, alignment constraints and stack layouts, is often implicit and not clear, and thus ABI should be explicitly and compactly (i.e., graphically) provided.

3.3.2 Design and Implementation

The code size of MieruCompiler is about 6,500 lines in C, GNU Flex and JavaScript (including JSON data files). JSON data files have all the data required by the MieruCompiler tooltips. As a GUI toolkit for Web browsers, we use ExtJS 2.2.

MieruCompiler is implemented as a rich internet application (RIA), which implies no installation procedure is required. Moreover, MieruCompiler is easy for the students to use, since it runs on Web browsers like Firefox and Google Chrome.

We needed to modify XCC so that MieruCompiler can obtain data being visualized from XCC. One important design policy is to minimize this modification to XCC, which is required because it becomes much difficult for the students to understand the XCC source code if XCC is heavily modified.

Although we cannot go into detail for the lack of space, we achieved this goal; only 5 manual source code instrumentations to XCC were required, whose total code size is only 15 lines. One way we took is to define a macro emit_code (Fig. 6, bottom) to transparently override the original function emit_code (Fig. 6, top). The function emit_code is just a wrapper function of printf with an extra parameter ast for debugging purpose. The macro emit_code is recursively defined, but this works well since the C standard guarantees to expand a recursive macro only once (6.10.3.4 in [22]).

4. PRELIMINARY EVALUATION

In this section, we discuss our experience using XCC and MieruCompiler in Tokyo Institute of Technology and Wakayama University. 15 students tried to implement (a part of) code generator for i386 architecture using XCC and MieruCompiler. The student skills vary widely from a student with no experience on compiler to a student with experience on developing PL/0 compiler. At the beginning of the projects, we gave the source code of XCC front-end to the students. XCC and MieruCompiler were well received by most our students.

4.1 Experience on XCC

The following are comments from the students.

- XCC is suitable for learning a foundation of i386 assembly language programming.
- It is good that after the projects XCC gives other advanced exercises for eager students like implementing optimizers (register allocation in particular) and their own front-end.
- XCC gave me a new point of view for programming through understanding XCC’s AST and compilation mechanism.
- It is not clear for me to what extent XCC’s semantic analysis is done in the given front-end. (eg., in XCC, a type analysis is done, but no analysis on L-value and R-value is done.)
- XCC is difficult for less experienced students. Simpler compiler projects might be better for them using a simpler source language with basic operators, control structures (eg., if and while) and no functions; and a simpler virtual machine code like P-code.

4.2 Experience on MieruCompiler

The following are comments from the students.

- “Horizontal slicing” in MieruCompiler enables me to quickly understand the corresponding i386 assembly code when pointing at a code fragment of XCC, and thus it is quite useful. Stack layouts shown in MieruCompiler are also useful.
• Tooltips are popped up in MieruCompiler when pointing at a code fragment of i386 assembly code. This immediately reminds me of its meaning. Thus the tooltips are quite useful since the i386 addressing mode is complex in particular.

• I did not use the visualization mechanism at all. (Note: he is a relatively experienced student.)

• More detailed information should be provided in MieruCompiler, for example, about all members in AST (\texttt{struct AST}) and types (\texttt{struct Type}).

• I want MieruCompiler to provide a comprehensive tutorial and reference for i386 assembly language programming, since tooltips are not enough for less experienced students to learn i386 from the scratch.

5. RELATED WORK

To our knowledge, there are no ones available on the Internet that have the same concept as XCC and MieruCompiler, although many educational compilers/visualizers have been proposed so far.

5.1 Educational compilers

Unexpectedly, there are few educational compilers that have all the following features, (1) which generate native code, (2) whose source language is a strict syntactic subset of real-world procedural programming languages like C, (3) which have an integrated visualization tool with “horizontal slicing.”

PL/0 [14] do not generate native code, and the source language is not a subset of real-world procedural languages. [15] generate native code, but their source languages are not a subset of real-world procedural languages. Conversely, [16]’s source language is a subset of real-world procedural languages, but they do not generate native code.

MiniJava [3], Cool [15] and Bantam Java [17] are educational compilers whose source languages are object-oriented. Although the features in object-oriented languages like information hiding or inheritance are very important, we believe that they are too difficult for undergraduate students to learn in a first compiler course.

All the above do not have an integrated visualization tool with “horizontal slicing.”

5.2 Visualization tools for compilers

Many visualization tools have been proposed so far, but most of them are point solutions and thus not integrated. For example, many tools have a specific feature like visualizing only parsing [5]AC AST [6]AC assembly code [7]AC symbol tables [8]AC optimizers [9]AC compiler behaviors [10, 11]AD.

Only the UW illustrated compiler [13] provides an integrated visualization for the scanner, parser, PL/0 source code, generated P-code, AST, the runtime stack, etc. [13] mentioned that “horizontal slicing” is important, but it is not implemented due to the following reason:

There is nothing intrinsically difficult about implementing this slicing but maintaining the maps between IRs, including the graphical display of the IRs, was too complex for our implementation.

To our knowledge, there are no visualization tools that provide “horizontal slicing” even at present, while MieruCompiler provides an integrated visualization with “horizontal slicing.”

6. CONCLUSION

This paper proposes a novel visualization tool called MieruCompiler and an educational compiler called XCC for undergraduate students. To reduce the learning cost of XCC, MieruCompiler provides visualizations for (1) “horizontal slicing”, (2) tooltips for native machine instructions, etc., (3) stack layouts, etc. XCC and MieruCompiler support native code generation for students in a first compiler course without sacrificing their learning experience. As a preliminary evaluation, MieruCompiler was used in two universities, which produced promising results.

7. REFERENCES


