CTD: A Compiler Translation Debugger Using Interspersed Code

Derrick Coetzee
University of California, Berkeley
Berkeley, CA USA
dcoetzee@eecs.berkeley.edu

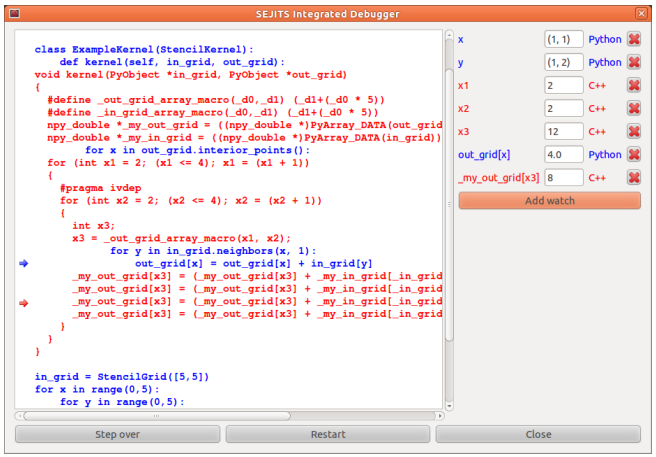

Figure 1: CTD interspersing Python and C++ while debugging a stencil program.

Abstract

As domain-specific languages multiply, more tool support is needed to support compiler production. One major obstacle is debugging incorrect compiler translations. We built an interface prototype for a debugger that intersperses high-level source code with the low-level target code emitted by the compiler. The two versions can be stepped through in tandem and data in both versions can be examined, allowing inconsistencies to be pinpointed. Preliminary evaluations with compilers written using the SEJITS framework show users prefer the new debugger interface, use the add watch feature more, and exhibit unique debugging behaviors when using the tool.

Keywords

Debugging, compilers, interspersed code

Introduction

Domain-specific languages (DSLs) enable programs in limited domains to be expressed at a very high level of abstraction, eliminating opportunities for error, facilitating communication with customers, and increasing opportunities for optimization. While low-performance domain-specific languages can rely on interpreters, high-performance or resource-constrained DSLs are generally compiled to a low-level, widely-supported efficiency language such as C or C++, requiring the

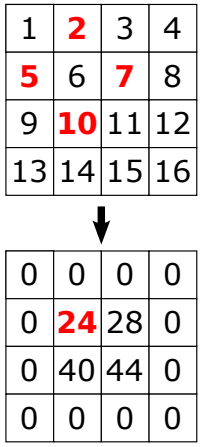

Figure 2: Stencil computation performed by first example program (see Evaluation section). Actual grid is 5 by 5.

production of a simple compiler. Because the number of potential domain-specific languages is large, these compilers must be made simple to write and debug for domain-expert developers without an extensive compiler background.

A number of measures can be taken to simplify the construction of these compilers: by embedding domain-specific languages in a subset of a general-purpose language, the parser phase is avoided and interoperability is simple. Toolkits like SEJITS further simplify the construction of these embedded domain-specific language compilers (called *specializers* for short) by providing common facilities like backend processing of C++ output and generic optimization. [1] However, while simplifying compiler construction eliminates some opportunities for errors, errors are still possible during the translation process, and these errors can be difficult to repair using standard debugging tools because they offer little insight into the complex relationship between input and output programs.

Our tool, Compiler Translation Debugger (CTD) effectively visualizes this relationship by interspersing input code, written in a domain-specific language embedded in Python, with generated output code, written in C++. Standard debugger commands, such as stepping and watching variables, are generalized to this setting by stepping through the two programs in tandem and watching different variables in both versions simultaneously.

In the remaining sections we will discuss related work, the design and features of CTD, the evaluation, and future work, respectively.

Related work

The technique is most immediately based on the Visual Studio IDE's *Disassembly Window*, [7] which intersperses C++ and its compiled assembly language equivalent (or .NET languages and MSIL bytecode). Like our tool, it also allows watching expressions in the target language (registers and memory locations). This feature is well-documented and widely-known, but has not been the subject of a user evaluation or other study, and does not support higher-level target languages like C/C++. It is not known whether it is used internally by Microsoft to debug problems with compiler translation, but it is regularly used by customers to offer insight into translation details.

A variety of techniques have been proposed for automatic compiler verification. Some, like Necula's proof-carrying code, [8] later developed into *typed assembly language* for compilers outputting native code, [2] seek to verify compilers by augmenting the compiler to propagate formal information about the source to annotations in the target code that can be easily and automatically verified. However, these place a daunting burden on the compiler writer, who needs effort and expertise to insert the necessary augmentations. Later work by Necula on translation validation was able to automatically verify semantic correctness of many compiler optimizations without the involvement of the optimizer. [9] This type of validator could in principle be supplied by the SEJITS framework, avoiding burdening the specializer writer, but requires knowledge about the intermediate form which is being transformed, which is currently determined entirely by the specializer.

A 2009 work by Leroy used interactive theorem provers to perform full semantic verification of a compiler for a

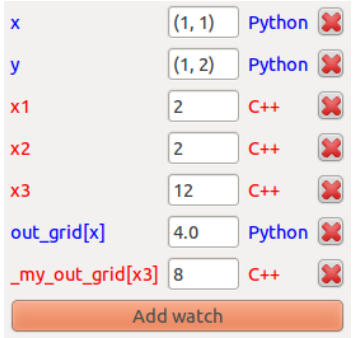

Figure 3: Detail of watch panel showing values of expressions in the Python and C++ program at the same point in execution. The difference between (x1, x2) and x points to an off-by-one bug.

```
            for y in in_grid.neighbors(x, 1):
⇒               out_grid[x] = out_grid[x] + in
      _my_out_grid[x3] = (_my_out_grid[x3] + _
      _my_out_grid[x3] = (_my_out_grid[x3] + _
⇒     _my_out_grid[x3] = (_my_out_grid[x3] + _
      _my_out_grid[x3] = (_my_out_grid[x3] + _
```

Figure 4: Detail of code view showing current point of execution in the Python (blue) and C++ (red) versions of the program. In this program, the Python loop has been unrolled, so the two lines of Python are run for each of the four lines of C++.

subset of C including optimizations. [6] While this technique is also interactive and achieves the ideal of full verification, use of interactive theorem provers requires a unique skill set and has very high overhead, requiring years to verify a compiler that takes only weeks to produce. While this method is far more practical for small compilers like those in SEJITS, it remains outside the skill set of typical domain experts.

Wu et al. have investigated debugging support for domain-specific languages (DSLs) in particular. Their DSL Debugging Framework (DDF), a set of Eclipse plug-ins, generates DSL support tools including debuggers from the DSL's declarative specification. [11] While this work illustrates how to effectively separate out debugging support like line number tracking in a DSL compiler using aspects, which is also needed in our scenario, they aim to provide only basic debugging support. This is not needed for embedded DSLs, which can rely on the host language tools for basic debugging support.

The LLVM Compiler Framework, widely used in research for prototyping compiler extensions, provides automated support for isolating test failures with its *code generator debugger* and *miscompilation debugger*, which use a simple delta debugging algorithm to narrow down the portion of the input causing the failure. [5] Work by Jeremy Singer on LLVM applies *concept assignment* to debugging of code generators, a technique for relating source regions to human-oriented concepts both automatically and manually, and then identifying inconsistencies. [10] It provides a source visualization that highlights inconsistencies, but does not show output code.

Although not targeted specifically at compiler translation, novel user interfaces for debugging have been proposed, most notably Ko et al's Whyline,[3] which aims to answer questions about why a certain unexpected result is produced by a program by tracing a behavior back to the program lines causing it. Although useful for many programs, in a typical compiler architecture a central data structure representing execution behavior is refined and altered by many successive optimization phases, which may lead to unwieldly sets of influencing code points.

Design and features

The debugger evaluated in this work is a standalone application with only basic functionality: a read-only display of interspersed Python and C++ code, colored by language, buttons to step and restart, and the ability to add and remove watches on any expression in either the Python or C++ version. Setting breakpoints is not supported, and because only a single function is being debugged, there is no distinction between step into/step over and no step out feature or visualization of the call stack.

Two arrows indicate the current position in both the C++ and Python source. The step button performs tandem stepping, so that these two arrows always refer as close as possible to corresponding points in execution. To enforce this, each step may step only the C++ version, only the Python version, or both depending on current state. The watch functionality hooks into gdb (GNU Debugger) and pdb (Python Debugger) back-ends and can examine arbitrary expressions.

Evaluation

Two example programs were prepared using the SEJITS stencil specializer, which performs computations on a grid where each output element in a function of nearby input elements (see Figure 4 for an example). The first was a simple one that adds the four surrounding elements, while

Example	CDT	C++
Simple	2 of 3	2 of 3
Bilateral	1 of 3	0 of 3

Figure 5: Number of subjects who completed each example.

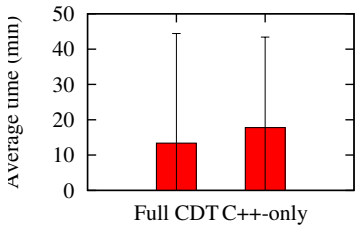

Figure 6: Average time to locate bug in first example. Bars indicate 95% mean confidence intervals.

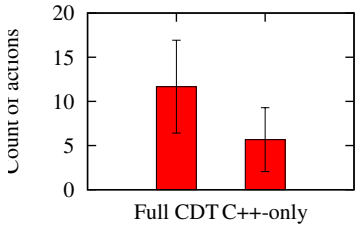

Figure 7: Count of uses of the "add watch" tool. Bars indicate 95% mean confidence intervals.

the second is a more complex one that performs a bilateral filter on an image. In each the default sequential C++ translation was used. A different deterministic error was introduced into the compiler code for each example: in the first example the loop bounds were off by one, and in the second a Manhattan distance function was used in place of the correct Euclidean distance function.

Subjects were each assigned to debug one program using the full version of CDT and the other program using a restricted C++-only version which merely interspersed Python code as C++ comments, representing the state of the art. Half the subjects used the full CDT on the first example and half on the second example. Subjects were informed there was an error in the translation process and asked to identify the error; they were given a brief explanation of available features in the debugger, but no explanation of the program. Debugging of each program ends after 20 minutes or if they successfully find the error and can clearly explain it (preliminary evaluation determined that a subject could successfully use CDT to identify each bug within 20 minutes). They were not able to examine the compiler source code. Every UI action and its time was logged, and subjects were encouraged to voice their thoughts, which were recorded. Immediately after each example, the error was explained, ensuring that all subjects entered the second example with the same knowledge. All subjects use the same PC in a lab environment. Following the debugging sessions, participants answer a short survey to collect subjective impressions and assess learning.

Six subjects were tested. Completion rates were very low on the second example, as shown in Figure 5, so we analyze time to completion only for the first example. A one-sided t-test shows no evidence of a difference in completion time ($p > 0.15$).

We also analyzed the number of UI actions performed by each participant. No evidence of a difference was observed between the two debuggers in the number of step actions (two-sided t-test, $p > 0.53$), but subjects used the "add watch" function significantly more when using the full CDT (two-sided t-test, $p < 0.04$), as shown in Figure 7, with a sample mean difference of 6 (population mean difference between 0.4 and 11.6 with 95% probability). We speculate this is because the comparative strategies used with the full CDT (see below) required more watches to perform. This was an overall trend and there was no significant difference in either example by itself (both $p > 0.20$).

In the survey portion, all subjects indicated a subjective preference for the full CDT debugger ("easier to recognize where in the code the error was occuring", "helpful to see exactly which python lines mapped to which c++ lines.") although one subject noted that more step actions were sometimes needed. All subjects successfully identified the meaning of UI elements like the red and blue text and the current position arrow. Subjects indicated average familiarity with debuggers in general (2.9/5), C++ debuggers (2.8/5), C++/Python interoperation (2.8/5), and the SEJITS platform (due to being drawn from the SEJITS developer pool; 3.1/5); but were unfamiliar with Python debuggers (1.4/5). Users were inconsistent with their descriptions of the program's behavior, ranging from detailed and accurate to vague and incomplete ("performing a calculation over the neighbors on a grid") to inaccurate, with more variation between users than between debuggers.

Subjective differences in how the two tools were used emerged during observation. When using the full CDT,

users added watches on corresponding expressions expected to have the same value (e.g. the current output element, certain subexpressions) and compared values during tandem stepping. Users using the full CDT also exploited the relative strengths and weaknesses of the backend debuggers (gdb could not easily print out array contents, while pdb could), and added Python watches to aid program understanding (e.g. "what does this function do?"). In many cases, even users who did not locate the bug narrowed down the line or region in which it was occurring. Users of the C++-only debugger were less strategic, watching local variables and stepping to observe program behavior, often with no clear goal in mind.

Limitations of evaluation

The pilot study exposed a number of critical limitations in our evaluation strategy that would influence the design of future studies of CTD. Most problematically, the scenario in which the debugging developer has no knowledge of the program under debug is unusual in practice, and in at least one case (debugging the bilateral filter example using the C++-only tool) made the bug completely impossible to identify. A more realistic study would give information to subjects regarding the design of both the compiler and the example being translated (in a controlled manner such as a design document or a scenario description), and would give access to the compiler and framework source code. This would also help to address the problem that even the most skilled subjects spent a large proportion of the allocated time understanding the program rather than using the debugger, which impacted completion rates.

The C++-only CDT also functioned as a highly artificial baseline. Many features of mature debuggers were absent; subjects said they would have considered using "breakpoints, especially conditional", "value of all local variables in current frame", "seeing entire Python/C++ arrays", and "back one step." CDT also has certain UI usability issues that frustrated users (e.g. inability to resize the watch pane). More fundamentally, the state-of-the-art is not that users exclusively debug compiler output using a C++ debugger tool – they also have the opportunity to separately debug the Python version using the Python debugger, to modify the application to observe changes in translation behavior, to modify the translation process to facilitate debugging, and to insert trace and assert statements into the code, which are translated into the target environment. An effective debugging framework should embrace and integrate these disparate approaches to interactive compiler translation debugging.

Future work

Although sufficiently developed for evaluation, presently CTD is not fully functional. The correspondence between input and output code lines is hard-coded, as is the information about which version to step at each point, which allows tandem stepping to remain in sync. Certain issues, like tandem stepping in the presence of control flow differences, are unresolved. A realistic implementation would need to resolve these issues.

CTD is primarily limited to debugging deterministic errors in sequential translations; tandem stepping is less meaningful in a context including parallelism. Concurrent work is investigating automatic tools for debugging nondeterministic errors. Once such an error is isolated, CTD could be used to effectively present the error and further debug it.

In the SEJITS framework, specializers are written as a pipeline of independent phases and code passes through multiple intermediate representations. An improved CTD

could help isolate the phase where the error occurs and gain further insight into the translation process by visualizing and tandem stepping through an arbitrary subset of intermediate forms. An obstacle is that most intermediate forms are stored as a tree data structure with no obvious concise textual representation.

Acknowledgements

This work was performed at the UC Berkeley Parallel Computing Laboratory (Par Lab), supported by DARPA (contract #FA8750-10-1-0191) and by the Universal Parallel Computing Research Centers (UPCRC) awards from Microsoft Corp. (Award #024263) and Intel Corp. (Award #024894), with matching funds from the UC Discovery Grant (#DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009.

[2] B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. Type-based verification of assembly language for compiler debugging. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, pages 91–102, New York, NY, USA, 2005. ACM.

[3] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[4] N. Kumar, B. R. Childers, and M. L. Soffa. Tdb: a source-level debugger for dynamically translated programs. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 123–132, New York, NY, USA, 2005. ACM.

[5] C. Lattner. LLVM bugpoint tool: design and usage, Documentation for the LLVM System. http://llvm.org/docs/Bugpoint.html#codegendebug, 2011.

[6] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[7] Microsoft. How to: Use the Disassembly Window, Visual Studio 2010, MSDN. http://msdn.microsoft.com/en-us/library/a3cwf295.aspx, 2011.

[8] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[9] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[10] J. Singer. Concept assignment as a debugging technique for code generators. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, pages 75 – 84, sept.-1 oct. 2005.

[11] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1370–1374, New York, NY, USA, 2005. ACM.