Interactive High-Performance Language Design and Implementation

Jonathan Bachrach

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

http://www.ai.mit.edu

The Problem: Can we combine the best of both scripting and delivery programming languages, where scripting languages support rapid prototyping and delivery languages produce small and fast applications? The aims of the two types of languages were thought to be largely irreconcilable.

Motivation: Traditionally there has been a split between programming languages designed for prototyping and those for delivery. Unfortunately, examining the software life-cycle more closely, one sees that a large percentage of cost and effort is spent after first deployment, including a large amount of time spent re-engineering the original code. If an application is first developed in a scripting language and then translated to a delivery language, much of the original design information is lost as well as the rapid prototyping tools that make reengineering so easy. Furthermore, there are many application domains (e.g., robotics and multimedia) where both performance and malleability are important, that is, it is extremely useful to interactively debug and modify optimized code. This suggests the need for a single language and implementation that supports interactive development of optimized code.

Previous Work: One of the earliest efforts towards achieving this goal was the Lisp machine [1] project. The solution was to design and implement specialized hardware to support efficient execution and live updating of a dynamic language. A more recent effort involved writing a virtual lisp machine which can run on stock hardware. Unfortunately, this system is unable to compete with delivery languages for the most demanding numeric tasks such as DSP.

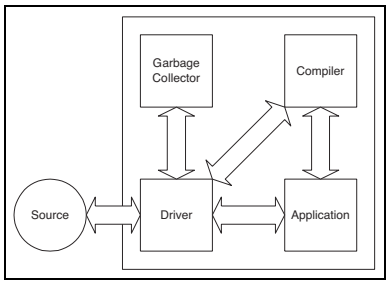
Dylan's approach [4] was to allow the programmer to seal performance critical parts of a program in order to permit partial optimizations. Although this allows the program to stay in the same language, this unfortunately leads to premature optimizations, forcing the programmer to decide which parts of a program would be frozen in the future and thereby having to trade off malleability for optimizeability.

One of the first attempts at dynamic compilation took place in the SELF system [2]. The implementation involved a sophisticated profile-guided optimizing compiler which ran along side the runtime. The runtime would discover opportunities for speed ups, and then proceed to aggressively compile those hotspots while tracking assumptions that, if violated, would require recompilation. Unfortunately, the SELF system did not address advanced issues of redefinition such as updating obsolete objects and multiple threads. Furthermore, their language design and implementation might not be appropriate for real-time applications. In particular, feedback-guided compilation might be too intrusive and might not provide acceptable guarantees nor control over optimization.

Java [3] popularized the notion of a Just In Time compiler which can give many of the benefits of whole program compilation in a lightweight fashion. Furthermore, Sun and others productized the SELF feedback-guided compilation ideas. Unfortunately, Java itself was not designed to be a friendly scripting / prototyping language. In particular, it is strongly typed, its redefinition semantics are ill-defined, and its definitions are at too coarse a granularity.

Approach: Our approach involves a new language called Proto, a new compiler architecture called *incremental whole program analysis*, and a redefinition-capable runtime. Proto is a prefix-syntaxed dynamic object-oriented language that is meant to be simple, powerful and extensible. Furthermore, it includes a powerful soft typing system that allows the inclusion of type declarations that can aid in the optimization of performance critical code. It also includes well-defined redefinition semantics, including the notion of versioned objects.

The implementation will include just-in-time whole program compilation, including a dependency tracking



scheme that will support full interactive redefinition of highly optimized code and objects. During compilation, dependencies are recorded against assumptions made. Recompilation is triggered when assumptions change.

The system will include a fast implementation of updating obsolete objects while maintaining fast object access. A variety of approaches will be explored. The first approach is to stop the world and precisely trace the image and update all references to obsolete objects to point to updated objects. Unfortunately, this might cause unacceptable delays. One approach for performing this update more lazily involves the notion of forwarding pointers implemented using standard virtual memory hardware. Other techniques will also be explored and their merits evaluated.

Finally, we will explore variations of whole-program optimizations that are both fast and incremental. The fastest algorithms tend to require a large amount of recompilation for even the smallest changes. We will explore variations that pay a slight overhead in order to support fast recompilation. For example, the fastest dynamic type check and method selection algorithms are both very sensitive to the class hierarchy, where any change to it requires a large recompilation. Our approach will involve variations on these algorithms which are less sensitive to such information without sacrificing much performance.

Impact: First, this work will provide a practical and useable system targetted towards interactive high-performance applications such as robotics, electronic music, ubiquitous computing, and server applications. Second, this work will contribute towards the understanding of language design and implementations that hit a sweet spot between the seemingly divergent concerns of interactive development and high performance.

Future Work: In short, we will be investigating ways to provide more abstraction and more optimization while minimizing latencies. In particular, we will investigate abstraction mechanisms that are sorely missed in typical high performance / real-time applications. One particularly promising direction is the Series construct for loopless programming [5]. This construct is particularly well suited to bulk operations common in DSP applications. In concert with this, we will also be looking at new optimization techniques to reduce the overhead of these abstractions. One important area will be practical partial evaluation. Finally, we will continue to develop new algorithms that minimize latencies both during redefinition and during execution.

Research Support: This research is supported in part by DARPA under the Express contract F30602-97-2-0013.

References:

[1] Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, and Daniel Weinreb. LISP machine progress report. Technical Memo AIM-444, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, August 1977.

[2] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototyypes. *ACM SIGPLAN Notices*, 24(10):49–70, October 1989. OOPSLA '89 Conference Proceedings, Norman Meyerowitz (editor), October 1989, New Orleans, Louisiana.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[4] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[5] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems,* 13(1):52–98, January 1991.