Benchmarking Typestate-Oriented Programming Languages

Benjamin W. Chung
Carnegie Mellon University
bwchung@andrew.cmu.edu

Abstract

The performance of typestate-oriented programming languages is difficult to evaluate as existing benchmarks do not exercise the unique features of these languages. We address this by developing a new benchmark suite specifically designed to evaluate typestate-oriented functionality. These benchmarks model projected applications, providing overhead and memory loads similar to actual applications.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Performance, Experimentation

Keywords Typestate, Protocols, Benchmarks, Performance, Dynamic Behavior

1. Introduction

Typestate-oriented programming languages raise the level of abstraction by directly expressing protocols, while maintaining a similar underlying structure to that of other dynamic languages. As dynamically modified interfaces are widely used in many applications, such as many Javascript applications and some virtual machines [2, 4], protocol alteration is a very important part of programming in dynamic languages. However, even in existing dynamic programming languages, interface alteration after object creation remains unexamined by many leading benchmark suites [5].

2. Plaid

Plaid is a new programming language being developed to support the idea of first-class state [6]. Each state can have fields and methods, and methods can transition the receiver. This enables several new design concepts that are little used in more traditional object-oriented languages.

```
state Socket {
  val identifier;
}
state ClosedSocket case of Socket {
  method open() {
    this <- OpenSocket;
  }
}
state OpenSocket case of Socket {
  method read() { ... }
  method write() { ... }
  method close() {
    this <- ClosedSocket;
  }
}
```

Figure 1. A basic Plaid program

An example of Plaid code can be found in Figure 1, which models a simple socket representation with two states, ClosedSocket and OpenSocket. Socket can transition between the two states via the open and close methods, and both expose custom functionality depending on the state they are in.

3. Existing Benchmarks

Existing benchmark suites are focused primarily on determining the speed of a system at performing certain algorithms, such as manipulating a splay tree[1]. It has been shown that these standard metrics are not representative of real-world application speed, and that optimizations targeting these benchmarks do not always improve speed of applications [1, 3]. In addition, little work has been done towards a benchmark suite for a typestate-oriented language such as Plaid. Two suites, DaCapo and V8, are particularly relevant for our project, as they represent commonly used benchmark suites for large programs and dynamic languages, respectively.

DaCapo DaCapo is a benchmark suite written to analyze the performance of Java Virtual Machines (JVMs) in actual use cases. It uses a selection of benchmarks that are common applications for the JVM, primarily based on large open-source software projects. This core causes the suite to represent the performance of a large number of Java applications,

SPLASH'12, October 19–26, 2012, Tucson, Arizona, USA.
ACM 978-1-4503-1563-0/12/10.

[1] http://v8.googlecode.com/svn/data/benchmarks/v7/run.html

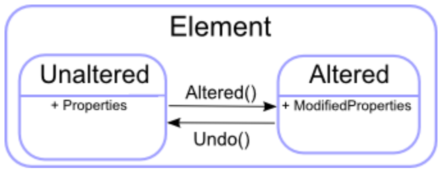

Figure 2. State diagram for implementation of the action pattern.

as many applications either use the frameworks that make up DaCapo or use similar algorithms and memory structures to them [1]. However, this same functionality makes it difficult to apply the DaCapo suite to the typestate-oriented model, as it is highly Java specific.

V8 The V8 benchmark suite was written by the V8 team to act as a measure of the speed of the V8 Javascript engine. The V8 benchmark uses algorithmic benchmarks to arrive at a composite number, such as the DeltaBlue constraint solver[2]. Despite this popularity, V8 has significant disadvantages, focusing primarily on mathematical speed, rather than dynamic functionality.

4. Design Goals

Benchmarks that accurately represent the performance of a real application are hard to create by using small programs that a single algorithms. To mitigate this issue, we will create programs that perform tasks that are similar to those that a actual user might need. These programs should run algorithms that are reasonable use cases for the language, and should run in a reasonable amount of time. The use of a typestate-oriented language also imposes some requirements on the type of benchmarks used, so benchmarks should use state change regularly to evaluate the performance of protocol dynamism in Plaid.

5. Proposed Benchmarks

We have several benchmarks planned for the suite, that analyze several categories of application. We will be using a basic transactional database as well as heavily modified traditional benchmarks, such as sorting benchmarks and a subset of V8. These will benchmark different aspects of a potential real application.

Transactional Database A transactional database has features that can be modeled easily via typestate, such as the underlying tree data structure and the actions on the database. This benchmark will preform a large number of sequential operations on a B-tree data structure. Typestate can be used to implement the transaction system via a modification state, which tracks whether a particular node in the B-tree has been modified. This system can allow rollbacks of a transaction simply by changing the state of the affected nodes within the tree. A diagram of this model can be seen in Figure 2.

[2] https://developers.google.com/v8/design

Benchmark	Plaid	Javascript	Java
Shell Sort	4.19	.131	.022
Binary Search Tree	1.09	.41	.0017
Splay Tree	18.26	.28	.11

Table 1. Preliminary comparison of execution times in seconds

Algorithmic We also plan on porting traditional benchmarks from the V8 suite into Plaid, to analyze the speed of the Plaid runtime at executing standard algorithms, despite the limitations discussed above. In addition, we have created two more benchmarks to provide simpler, garbage collector intense operations. This reuse of traditional benchmarks also enables comparison of Plaid to the original language. In many cases, the more traditional benchmarks can be modified to take advantage of typestate oriented functionality. The Richards and BST benchmarks both use state change extensively. This is caused by the prevalence of state and state-like patterns in traditional object-oriented code.

6. Results

Current Benchmarks At the present time, we only have a small subset of the V8 suite, as well as our new algorithmic benchmarks. Javascript versions of all of our current benchmarks exist, allowing us to compare the existing performance of Plaid to V8.

Results Using our preliminary benchmarks, we have determined that Plaid is approximately 1 order of magnitude slower than the equivalent JavaScript program, and 2 orders slower than the equivalent Java program, as seen in table 1. All of the preliminary benchmarks were highly algorithmically centered, with large memory usage and extreme repetition and recursion. Results were gathered on a computer running the 1.6 JRE on Windows 7 with a Intel Core i7-2760QM CPU at 2.40GHz, and 8.00 GB DDR3 PC3-10600 RAM.

References

[1] S. M. Blackburn, R. Garner, B. Wiedermann, and et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*.

[2] A. Gal, B. Eich, M. Shaver, D. Anderson, and et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM SIGPLAN 2009*, PLDI '09.

[3] M. Maass and I. Shafer. Instrumenting V8 to measure the efficacy of dynamic optimizations on production code. 2012.

[4] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *ACM SIGPLAN 2010*, PLDI '10, 2010.

[5] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. *SIGPLAN Not.*, 2011.

[6] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in plaid. In *OOPSLA '11*.