

Compilation Techniques for Customizing Large Java Libraries

Teck Tok and Calvin Lin
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

In collaboration with Bill Alexander and Weiming Gu
IBM e-boss System Performance
Austin, TX

1 Area of Technical Interest

Compilers, performance analysis, languages.

2 Project Description

This project addresses the performance problems associated with the construction of large, complex software. We propose to develop compilers and compiler-based tools that leverage domain-specific information to improve runtime performance. We will develop our techniques for Java programs, and we will evaluate our techniques by applying them to clean-room implementations of Java class libraries.

The technical problem is the debilitating effects that software structuring techniques tend to have on runtime performance. Our specific solution is a compiler that customizes a single module implementation for many different client needs. Our solution increases information flow across interfaces by using annotations to describe domain-specific information abstractly in terms of dataflow properties; dataflow analysis is then used to push such information across interface boundaries. This information can be used to relax worst-case assumptions so that specialized routines can be used where appropriate, and this information can be used to restructure code so that resource management is not tied to module boundaries.

The general approach is a problem-driven one. We will iteratively identify common performance problems, devise mechanisms for solving them, and then evaluate our solutions. Interaction with IBM employees and their expertise is essential, so we propose that the PI spend time at IBM to work with the Java library and its applications.

The deliverables include the following:

- An analysis of performance bottlenecks in Java class libraries.
- An analysis of common solutions to these performance bottlenecks.

- A source-to-source translator for Java and possibly derivative tools such as semi-automated translators. These tools will encapsulate the technical advances (new optimizations and analyses) developed by this work.
- Source code for any optimized Java class libraries code.
- Source code for any annotations used in optimizing the Java class libraries.
- Technical papers describing our results.

3 Objectives and Goals

The goals of this project are to analyze the performance of the Java class libraries and to develop compiler-based techniques for improving its performance. Our subgoals are to identify performance problems, to understand their causes, to create general solutions to these problems, and finally to evaluate our solutions using both microbenchmarks and a more realistic Java workloads.

4 Long Term Impact on IT Industry and IBM

One of the biggest problems facing the information technology industry is the huge demand for quality software. The proposed project offers specific methods of reducing the complexity of software. In particular, the work makes existing software abstraction mechanisms—libraries, modules, layers, classes, and monitors—more efficient, more general, and more usable. Thus, the proposed work promises to improve programmer productivity and to have wide impact on both IBM and the computing industry.

5 Evaluation of Success

The primary technical metrics for success are runtime performance of Java class libraries and their application programs, the code size of the optimized Java class libraries, and the compile time for performing our optimizations. We will apply these metrics to a Java workload that is identified with IBM's help. Certain micro-benchmarks will likely also be used to help analyze and evaluate performance.

6 Technical Rationale

There are several impediments to obtaining good performance from large, complex software. Their sheer size implies a multi-programmer effort that makes it difficult to guarantee consistent code quality. More fundamentally, large software must be structured into modules, layers, or libraries to make it manageable. From our point of view, modules, layers and libraries are all essentially identical, as they share the same fundamental characteristics: They present a functional interface, and they hide implementation details from their clients. Unfortunately, these structuring techniques introduce several closely related performance problems:

1. **Different clients have different needs.** An implementation that is appropriate for one client can be inappropriate for another. For example, one IBM customer complained of poor performance when using the Java AWT to manipulate some black and white images.¹ It turned out that, unbeknownst to the customer, the AWT routine he was invoking was performing Floyd-Warshall dithering on the input image, which turned out to be a very costly no-op when applied to black and white images.
2. **“Separation of concerns” inhibits information flow across interfaces.** The performance of a module can typically be improved if the implementor is made aware of the client’s needs. For example, a routine that is used to render non-overlapping objects would be implemented very differently than one that is used to render overlapping objects, as the latter would sort the objects to render them in the correct order. But relying on such information would violate Parnas’ modularity principle, so such performance-related information does not typically flow across interfaces.
3. **Worst case assumptions provide generality at the expense of performance.** To provide correct behavior in all situations, libraries tend to make worst case assumptions. For example, library routines typically check parameters for validity. Such routines could be optimized if they could optimistically assume that all parameters were valid. Of course, the key in such cases would then be to ensure that optimistic, or unsafe, routines are only used where the assumptions are in fact correct. There are numerous other examples of this problem, as module implementors will perform excess copying of data, excess synchronization, and excess initialization of data, all to be conservatively safe. Other factors can contribute to these excesses as well. For instance, programmers who face deadline pressure or who do not fully understand how certain functions work can be tempted to copy data to limit the effects of their changes.
4. **Module structure leads to poor resource management.** To provide encapsulation and safety, memory management is often performed by lower level routines. However, resource management can often be improved by taking a global view. For example, if a routine `f()` needs to use a temporary buffer, it will naturally allocate memory for this buffer. However, if this routine is invoked repeatedly with fixed-sized buffers, a more efficient solution would allocate a single temporary buffer that all of these instances of `f()` could use. The optimized solution, shown below, reduces both memory allocation costs and garbage collection costs.

```
// Original Code: encapsulated resource management

    for (i=0; i<n; i++)
    {
        f(data[i]);      // f() will allocate memory on each invocation
    }

// Optimized Code: global resource management

    Buffer b = new Buffer(data[0].size()); // Extracted from within f()
    for (i=0; i<n; i++)
```

¹Thanks to Weiming Gu and Robert Reynolds for this anecdote.

```
{
    f'(data[i], b); // f'() does not allocate memory, so we reduce
}                // both memory allocation and garbage collection
```

Our proposed solution to these problems uses a compiler to customize a single module implementation for many different client needs. Two points about our solution are noteworthy. First, our solution preserves existing functional interfaces. Thus, from the programmer's point of view, the structuring benefits of modules and libraries are retained. Second, domain-specific information is critical. Examples of domain-specific questions that can be used to improve performance abound: Has the visibility of this widget been modified? Is this image black and white? Has this file been opened for writing? Is this number the address of a remote name service?

We have shown that our ideas work in the context of the C programming language and the PLAPACK parallel linear algebra library [1, 2, 3], and we now propose to develop infrastructure and techniques for applying these ideas to Java class libraries.

Java provides several new opportunities and challenges. First, we believe that Java programs will benefit more from our techniques because Java provides more mechanisms for encapsulating data behind interfaces, and thus encourages more inefficiencies. Second, Java's dynamic binding inhibits static analysis, so a heavier emphasis on runtime analysis will be needed. Third, Java's class hierarchies present opportunities for inheriting annotations from superclasses and for associating annotations with Java interfaces. These mechanisms should allow annotations to be reused just as classes and interfaces can be reused. Finally, Java's pointer semantics simplify pointer alias analysis.

References

- [1] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, pages 39–52, October 1999.
- [2] Samuel Z. Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*, August 2000.
- [3] Robert van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. The MIT Press, 1997.