# Customization of Java Library Classes using Type Constraints and Profile Information

Bjorn De Sutter[1], Frank Tip[2], and Julian Dolby[2]

[1] Ghent University, Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41 9000 Gent, Belgium
brdsutte@elis.ugent.be
[2] IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598
{ ftip,dolby }@us.ibm.com

**Abstract.** The use of class libraries increases programmer productivity by allowing programmers to focus on the functionality unique to their application. However, library classes are generally designed with some typical usage pattern in mind, and performance may be suboptimal if the actual usage differs. We present an approach for rewriting applications to use customized versions of library classes that are generated using a combination of static analysis and profile information. Type constraints are used to determine where customized classes may be used, and profile information is used to determine where customization is likely to be profitable. We applied this approach to a number of Java applications by customizing various standard container classes and the omnipresent StringBuffer class, and measured speedups up to 78% and memory footprint reductions up to 46%. The increase in application size due to the added custom classes is limited to 12% for all but the smallest programs.

## 1 Introduction

The availability of a large library of standardized classes is an important reason for Java's popularity as a programming language. The use of class libraries improves programmer productivity by allowing programmers to focus on the aspects that are unique to their application without being burdened with the unexciting task of building (and debugging!) standard infrastructure. However, library classes are often designed and implemented with some typical usage pattern in mind. If the actual use of a library class by an application differs substantially from this typical usage pattern, performance may be suboptimal.

Consider, for example, the implementation of the container classes such as `Vector` and `Hashtable` in package `java.util`. In designing the implementation of these containers, a large number of accesses to objects stored therein was (implicitly) assumed. Therefore, the allocation of auxiliary data structures encapsulated by the container (e.g., a `Vector`'s underlying array, or a `Hashtable`'s embedded array of hash-buckets) is performed *eagerly* upon construction of the container itself. This approach has the advantage that the container's access methods can assume that these auxiliary data structures have been allocated.