RC23037 (W0312-095) December 16, 2003
Computer Science

IBM Research Report

Customization of Java Library Classes Using Type Constraints and Profile Information

Bjorn De Sutter
Ghent University
Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41
9000 Gent, Belgium

Frank Tip, Julian Dolby
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

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.

However, as we shall see in Section 6, it is not uncommon for programs to create large numbers of containers that remain empty or that contain only small numbers of objects. In such cases, *lazy* allocation is preferable, despite the fact that the access methods become slower because they have to check if the auxiliary data structures have been allocated and create them if this is not the case.

Library classes may also induce unnecessary overhead if an application does not use all of the provided functionality. For example, most iterators provided by containers such as `Hashtable` are designed to be fail-fast (i.e., an exception is thrown when an attempt is made to use an iterator and a concurrent modification of its underlying container is detected). Fail-fast iterators are implemented by keeping track of the "version" of the container that an iterator is associated with, and incrementing a container's version number upon each modification. This "bookkeeping code" is executed, and space for its data is reserved, regardless of the fact whether or not iterators are used. For clients that do not use iterators, a customized container without iteration support can improve performance.

A third common case of unnecessary overhead occurs when single-threaded applications use library classes that are designed with multi-threaded clients in mind. For example, many Java programs frequently concatenate strings via calls to the `synchronized` method `java.lang.StringBuffer.append()`[3]. This means that a lock must be acquired for each call to this method, which is unnecessary for single-threaded applications. Performance can improved in such cases by rewriting the application to use custom, unsynchronized `StringBuffer`s.

We present an approach for automatically generating customized versions of Java library classes. Type constraints are used to determine where library classes can be replaced with custom versions without affecting type correctness or program behavior. Static analysis is used to determine situations where library functionality and synchronization can be removed safely. Profile information is used to obtain the usage characteristics of the created library class objects, and to determine where the use of custom library classes is likely to be profitable. Then, custom library classes are generated automatically from a template, and the bytecode of the client application is rewritten to use these custom classes in a way that is completely transparent to the programmer.

We applied these techniques to a set of benchmark Java applications, and customized various container classes in `java.util`, as well as class `StringBuffer`. We measured speedups ranging from -5 to 78% (19-24% on average, depending on the VM) and memory footprint reductions ranging from -1 to 46% (averaging 12%). Moreover, the addition of custom classes to the benchmarks resulted in only a modest increase in application size: less than 12% for all but the smallest programs.

The remainder of this paper is structured as follows. Section 2 presents a motivating example. Sections 3 and 4 present type constraints and their use for determining where custom classes may be used, respectively. Section 5 discusses the steps involved in profiling and generating custom classes. In Section 6, an

[3] Java compilers translate uses of the `+`-operator on `String` objects in Java into calls to `StringBuffer.append()`.

evaluation of our techniques on a set of benchmarks is presented. Sections 7 and 8 present related work, and conclusions and future work, respectively.

2 Motivating Example

We will use an example program that creates several `Hashtable` objects to illustrate the issues that arise when replacing references to standard library classes with references to custom classes. Class `Hashtable` is well-suited to serve as a motivating example because: (i) it is part of a complex class hierarchy in which it has both supertypes (e.g., `Map` and `Dictionary`) and subtypes (`Properties`), (ii) several orthogonal optimizations can be applied when creating custom hashtables[4], and (iii) `Hashtables` are heavily used by many (legacy) Java applications.

In order to create a customized version of, say, a `Hashtable`, one could simply create a class `CustomHashtable` that extends `Hashtable` and overrides some of its methods. Unfortunately, this approach has significant limitations. In particular, fully lazy allocation is impossible because `Hashtable`'s constructors always allocate certain auxiliary datastructures (e.g., an array of hash-buckets)[5], and each constructor of `CustomHashtable` must invoke one of `Hashtable`'s constructors. Moreover, each `CustomHashtable` object contains all of `Hashtable`'s instance fields, which introduces unneccessary overhead if these fields are unused (as was the case in the example discussed above of redundant iterator-related bookkeeping code). Therefore, customized classes will be introduced in a separate branch of the class hierarchy, as is indicated in Figure 1. The figure shows the standard library types `Hashtable`, `Map`, `Dictionary`, and `Properties`, and the inheritance relationships between them. Also shown are custom container classes `CachingHashtable` and `LazyAllocHashtable`, and an abstract class `AbstractCustomHashtable` that serves as a common superclass of customized versions of `Hashtable`.

We will use the example program shown in Figure 2 to illustrate the issues that arise when introducing custom classes such as those shown in Figure 1. This program creates a number of container objects, and performs some method calls on these objects. Observe that the program contains three allocation sites of type `Hashtable` and one of type `String` that we will refer as `H1`, `H2`, `H3`, and `S1`, as is indicated in Figure 2 using comments. We will now examine a number of issues that arise when updating allocation sites to refer to custom types.

Calls to methods in external classes. Allocation site `H1` cannot be updated to refer to a custom type because the object allocated at `H1` is passed to a constructor of `javax.swing.JTree` that expects an argument

[4] One particular optimization that can be applied to custom versions of class `Hashtable` is the removal of synchronization in cases where the client application is single-threaded. To avoid overhead in such cases, modern Java applications tend to use the similar class `HashMap`, in which the methods are not synchronized.

[5] While it is possible to specify the initial size of this array of hash-buckets upon construction of a `Hashtable`-object, the construction of this array-object cannot be avoided altogether if `CustomHashtable` is a subclass of `Hashtable`.

of type `java.util.Hashtable`. Since the code in class `JTree` is not under our control, the type of the parameter of `JTree`'s constructor must remain `java.util.Hashtable`, which implies that the types of `h1` and `H1` must remain `java.util.Hashtable` as well. Similar issues arise for calls to library methods whose return type is a *concrete*[6] standard container, such as the call to `System.getProperties()` on line 11, which returns an object of type `java.util.Properties`, a subtype of `java.util.Hashtable`.

Preserving type-correctness. Allocation sites `H2` and `H3` may be updated to refer to, for example, type `CachingHashtable`. However, if we make this change, we must also update `h2` and `h3` to refer to a superclass of `CachingHashtable` (i.e., `CachingHashtable`, `AbstractCustomHashtable`, `Map`, `Dictionary`, or `Object`) because the assignments on lines 5 and 6 would otherwise not be type-correct. The method calls `h2.put("FOO", "BAR")` and `h2.putAll(c)` impose the additional requirement that the `put()` and `putAll()` methods must be visible in the type of `h2`, and hence that `h2`'s type must be `CachingHashtable`, `AbstractCustomHashtable`, or `Map`. Furthermore, the assignment `h2 = h3` is only type-correct if the type of `h2` is the same as or a supertype of the type of `h3`, and the assignments `Properties p1 = System.getProperties()` and `h2 = p1` require that the type of `h2` must be a supertype of `java.util.Properties`. Combining all of these requirements, we infer that allocation sites `H2` and `H3` can only be updated to allocate `CachingHashtable` objects if both `h2` and `h3` are declared to be of type `Map`.

Preserving the behavior of casts. The cast expression on line 16 (indicated as `C1`) presents another interesting case. Observe that only objects allocated at sites `H3` and `S1` may be bound to parameter `o`. In the transformed program, the cast expression must succeed and fail in exactly the same cases as before. In this case, if the type of the object allocated at site `H3` is changed to `CachingHashtable`, changing the type of the cast to, for example, `AbstractCustomHashtable` will preserve the behavior of the cast (it will still succeed when parameter `o` points to an object allocated at site `H3` and it will still fail when `o` points to an object allocated at site `S1`). Furthermore, the assignment `Hashtable h4 = (Hashtable)o` is only type-correct if the type of `h4` is a supertype of the type referenced in the cast expression, and the method call `h4.contains(···)` implies that `h4`'s type must define the `contains(···)` method (in other words, `h4` must have type `AbstractCustomHashtable` or a subtype thereof). We conclude from the above discussion that having the cast refer to type `AbstractCustomHashtable` and declaring `h4` to be of type `AbstractCustomHashtable` is a valid solution[7].

[6] The use of *concrete* container types such as `Hashtable` (as opposed to *abstract* container types such as `Map`) in the signature of public library methods is often an indication of poor design, because it unnecessarily exposes implementation details. Nonetheless, this practice is pervasive. We counted 165 public methods in the JDK 1.3.1 standard libraries whose signature refers to `Hashtable`, `Vector`, or `Properties`.

[7] Several other solutions exist. For example, variable `h4` and cast `C1` can both receive type `CachingHashtable`.

The lessons learned from the above example can be summarized as follows: The customized program must satisfy *interface-compatibility constraints* that are due to the exchange of standard container objects with third-party libraries and *type-correctness constraints* implied by program constructs such as assignments that constrain the types of their subexpressions. Moreover, *run-time behavior* must be preserved for casts and `instanceof` operations.

Lines 5b, 15b, and 16b in Figure 2 indicate how allocation sites, declarations, and cast expressions in the original example program have been replaced with references to custom classes in accordance with the requirements that we discussed informally in this section. We will now turn our attention to a more precise treatment of these requirements.

3 Type Constraints

We will use an existing framework of *type constraints* [12, 17] to determine where the types of allocation sites and declarations can be updated to refer to custom types in a way that preserves the program's type-correctness and behavior. For each program construct, one or more type constraints express the subtype-relationships that must exist between the declared types of the construct's constituent expressions, in order for that program construct to be type-correct. By definition, a program is *type-correct* if the type constraints for all constructs in that program are satisfied.

In the remainder of this paper, we assume that the original program is type-correct. Moreover, we assume that the original program does not contain any up-casts (i.e., casts $(C)E$ in which the type of expression E is a subclass of C). This latter assumption is not a restriction, as there is no need for up-casts in Java byte code[8].

3.1 Notation and Terminology

Following [17], we use the term *declaration element* to refer to declarations of local variables, parameters in static, instance, and constructor methods, fields, and method return types, and to type references in cast expressions. In what follows, v, v' denote variables, M, M' denote methods, F, F' denote fields, C, C' denote classes, I, I' denote interfaces, and T, T' denote types[9]. It is important to note that the symbol M denotes a method together with all its signature and return type information and the reference to its declaring type. Similarly, F and C denote a field and a type, respectively, together with its name, type in which it is declared and, in the case of fields, its declared type.

Moreover, the notation E, E' will be used to denote an expression or declaration element at a specific point in the program, corresponding to a specific node

[8] In Java source code, up-casts are sometimes needed for explicit resolution of overloaded methods.

[9] In this paper, the term *type* will denote a class or an interface.

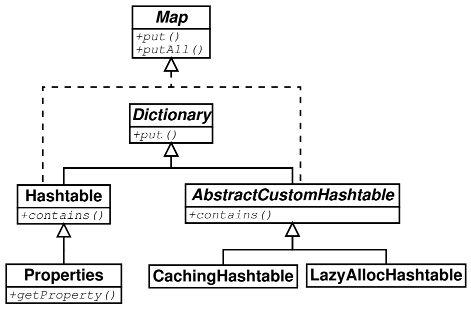

Fig. 1. A fragment of the standard container hierarchy in package `java.util.*`, augmented with several customized containers. The figure shows, for a number of relevant methods, the most general type in which they are declared.

```
1      public class Example {
2        public static void foo(Map m) {
3          Hashtable h1 = new Hashtable();  /* H1 */
4          JTree tree = new JTree(h1);
5          Hashtable h2 = new Hashtable();  /* H2 */
5b         Map h2 = new CachingHashtable();
6          Hashtable h3 = new Hashtable();  /* H3 */
6b         Map h3 = new CachingHashtable();
7          bar(h3);
8          h2 = h3;
9          h2.put("FOO", "BAR");
10         h2.putAll(m);
11         Properties p1 = System.getProperties();
12         String s = p1.getProperty("java.class.path");
13         h2 = p1;
14       }
15       public static void bar(Object o) {
16         Hashtable h4 = (Hashtable)o;     /* C1 */
16b        AbstractCustomHashtable h4 = (AbstractCustomHashtable)o;
17         if (h4.contains("FOO")){ ... }
18       }
19       public static void bad(){
20         String s = new String("bad");    /* S1 */
21         bar(s);
22       }
23     }
```

Fig. 2. This figure shows an example program P_1, which consists of lines 1–23 excluding lines 5b, 6b, and 16b. A customized version of this program that uses the custom class hierarchy of Figure 1 can be obtained from P_1 by replacing lines 5, 6, and 16, with lines 5b, 6b, and 16b, respectively.

in the program's abstract syntax tree. We will assume that type information about expressions and declaration elements is available from the compiler.

A method M is *virtual* if M is not a constructor, M is not private and M is not static. Definition 1 defines the concept of *overriding*[10] for virtual methods.

Definition 1 (overriding). *A virtual method M in type C overrides a virtual method M' in type B if M and M' have identical signatures and C is equal to B or C is a subtype of B. In this case, we also say that M' is overridden by M.*

Definition 2 defines, for a given method M, the set *RootDefs*(M) of methods M' that are overridden by M that do not override any methods except for themselves. Since we assume the original program to be type-correct, this set is guaranteed to be non-empty. For example, in the standard collection hierarchy, we have that *RootDefs*(`Hashtable.put()`) = {`Map.put()`,`Dictionary.put()`} because `Map` and `Dictionary` are the most general types that declare `put()` methods that are overridden by `Hashtable.put()`.

Definition 2 (RootDefs). *Let M be a method. Define:*

$$\mathit{RootDefs}(M) = \{\ M' \mid M \textit{ overrides } M', \textit{ and there exists no } M''\ (M'' \neq M') \textit{ such that } M' \textit{ overrides } M''\ \}$$

Figure 3 shows the notation used to express type constraints. A *constraint variable* α is one of the following: T (a type constant), $[E]$ (representing the type of an expression or declaration element E), *Decl*(M) (representing the type in which method M is declared), or *Decl*(F) (representing the type in which field F is declared). A *type constraint* is a relationship between two or more constraint variables that must hold in order for a program to be type-correct. In this paper, a *type constraint* has one of the following forms: (i) $\alpha_1 \triangleq \alpha_2$, indicating that α_1 is defined to be the same as α_2 (ii) $\alpha_1 \leq \alpha_2$, indicating that α_1 must be equal to or be a subtype of α_2, (iii) $\alpha_1 = \alpha_2$, indicating that $\alpha_1 \leq \alpha_2$ and $\alpha_2 \leq \alpha_1$, (iv) $\alpha_1 < \alpha_2$, indicating that $\alpha_1 \leq \alpha_2$ but not $\alpha_2 \leq \alpha_1$, (v) $\alpha_1^L \leq \alpha_1^R$ **or** $\cdots$ **or** $\alpha_k^L \leq \alpha_k^R$, indicating that $\alpha_j^L \leq \alpha_j^R$ must hold for at least one j, $1 \leq j \leq k$, (vi) $\alpha_1 \not\leq \alpha_2$, indicating that α_1 must not be equal to or be a subtype of α_2

In discussions about types and subtype-relationships that occur in a specific program P, we will use the notation of Figure 3 with subscript P. For example, $[E]_P$ denotes the type of expression E in program P, and $T' \leq_P T$ denotes a subtype-relationship that occurs in program P. In cases where the program under consideration is unambiguous, we will frequently omit these P-subscripts.

3.2 Inferring Type Constraints

We will now present the rules that will be used for inferring type constraints from various Java constructs and analysis facts.

[10] Note that, according to Definition 1, a virtual method overrides itself.

$[E]$	the type of expression or declaration element E
$[M]$	the declared return type of method M
$[F]$	the declared type of field F
$Decl(M)$	the type that contains method M
$Decl(F)$	the type that contains field F
$Param(M, i)$	the i-th formal parameter of method M
$T' \leq T$	T' is equal to T, or T' is a subtype of T
$T' < T$	T' is a proper subtype of T (i.e., $T' \leq T$ and not $T \leq T'$)

Fig. 3. Type constraint notation.

program construct(s)/analysis fact(s)	**implied type constraint(s)**	
assignment $E_1 = E_2$	$[E_2] \leq [E_1]$	(1)
method call $E.m(E_1, \cdots, E_n)$ to a virtual method M	$[E.m(E_1, \cdots, E_n)] \triangleq [M]$	(2)
	$[E_i] \leq [Param(M, i)]$	(3)
	$[E] \leq Decl(M_1)$ **or** $\cdots$ **or** $[E] \leq Decl(M_k)$ where $RootDefs(M) = \{ M_1, \cdots, M_k \}$	(4)
access $E.f$ to field F	$[E.f] \triangleq [F]$	(5)
	$[E] \leq Decl(F)$	(6)
`return` E in method M	$[E] \leq [M]$	(7)
constructor call `new` $C(E_1, \cdots, E_n)$ to constructor M	$[E_i] \leq [Param(M, i)]$	(8)
direct call $E.m(E_1, \cdots, E_n)$ to method M	$[E.m(E_1, \cdots, E_n)] \triangleq [M]$	(9)
	$[E_i] \leq [Param(M, i)]$	(10)
	$[E] \leq Decl(M)$	(11)
cast $(C)E$	$[(C)E] \leq [E]$ if $[E]$ is a class	(12)
for every type T	$T \leq$ `java.lang.Object`	(13)
	$[$`null`$] \leq$ `T`	(14)
implicit declaration of `this` in method M	$[$`this`$] \triangleq Decl(M)$	(15)
declaration of method M (declared in type T)	$Decl(M) \triangleq T$	(16)
declaration of field F (declared in type T)	$Decl(F) \triangleq T$	(17)
explicit declaration of variable or method parameter $T\ v$	$[v] \triangleq T$	(18)
declaration of method M with return type T	$[M] \triangleq T$	(19)
declaration of field F with type T	$[F] \triangleq T$	(20)
cast $(T)E$	$[(T)E] \triangleq T$	(21)
expression `new` $C(E_1, \cdots, E_n)$	$[$ `new` $C(E_1, \cdots, E_n)] \triangleq C$	(22)
M' overrides M, $M' \neq M$	$[Param(M', i)] = [Param(M, i)]$	(23)
	$[M'] = [M]$	(24)
for each cast expression $(C)E$, and each allocation expression $E' \in PointsTo(P, E)$ such that $[E']_P \leq [(C)E]_P$	$[E'] \leq [(C)E]$	(25)
for each cast expression $(C)E$, and each allocation expression $E' \in PointsTo(P, E)$ such that $[E']_P \not\leq [(C)E]_P$	$[E'] \not\leq [(C)E]$	(26)
expression E that occurs in the libraries such that $[E]_P = T$	$[E] = T$	(27)

Fig. 4. Type constraints for a set of core Java language features. Rules (1)–(17) define the types of expressions and impose constraints between the types of expressions and declaration elements. Rules (18)–(22) define the types of declaration elements and allocation expressions in the original program. Rules (23)–(26) show additional type constraints that are needed for the preservation of program semantics. Rule (27) shows an additional type constraint needed to preserve interface-compatibility.

Type-Correctness Constraints Rules (1)–(17) in Figure 4 shows the type constraints that are implied by a number of common Java program constructs. For example, constraint (1) states that an assignment $E_1 = E_2$ is type correct if the type of E_2 is the same as or a subtype of the type of E_1. For a virtual method call $E.m(E_1, \cdots, E_n)$ that statically resolves to a method M, we define the type of the call-expression to be the same as M's return type (rule (2)), and we require that the type of each actual parameter E_i must be the same as or a subtype of the type of the corresponding formal parameter $\mathit{Param}(M, i)$ (rule (3)). Moreover, a declaration of a method with the same signature as M must occur in a supertype of the type of E. This latter fact is expressed in rule (4) using Definition 2 by way of an **or**-constraint. For cast expressions of the form $(C)E$, rule (21) defines the type of the cast to be C. Moreover, rule (12) states the requirement that the type of E must be a supertype of C[11].

$\mathit{PointsTo}(P_1, \texttt{h1}) = \{\, \texttt{H1} \,\}$	$\mathit{PointsTo}(P_1, \texttt{o}) = \{\, \texttt{H3}, \texttt{S1} \,\}$
$\mathit{PointsTo}(P_1, \texttt{h2}) = \{\, \texttt{H2}, \texttt{H3}, \texttt{P1} \,\}$	$\mathit{PointsTo}(P_1, \texttt{p}) = \{\, \texttt{P1} \,\}$
$\mathit{PointsTo}(P_1, \texttt{h3}) = \{\, \texttt{H3} \,\}$	$\mathit{PointsTo}(P_1, \texttt{s}) = \{\, \texttt{S1} \,\}$
$\mathit{PointsTo}(P_1, \texttt{h4}) = \{\, \texttt{H3} \,\}$	

Fig. 5. Points-to information for program P_1 of Figure 2, as computed using a variation on the flow-insensitive, context-insensitive 0-CFA algorithm [11] that propagates allocation sites rather than types. Here, `P1` represents the allocation site(s) at which the `Properties` objects returned by `System.getProperties()` are allocated.

Rules (18)–(22) define the types of variables, parameters, fields, method return types, casts, and allocation sites in the original program.

Behavior-Preserving Constraints The type constraints discussed thus far are only concerned with type-correctness. In general, additional constraints must be imposed to ensure that program behavior is preserved. Rules (23) and (24) state that overriding relationships that occur in the original program P must also occur in the rewritten program P'.

Rules (25) and (26) state that the execution behavior of a cast $(C)E$ must be preserved. Here, the notation $\mathit{PointsTo}(P, E)$ refers to be the set of objects (identified by their allocation sites) that an expression E in program P may point to. Any of several existing algorithms [14, 8, 15] can be used to compute this information. Figure 5 shows the points-to information that will be used in the examples below. Rule (25) ensures that for each E' in the points-to set of E for which the cast succeeds, the cast will still succeed in P'. Likewise, Rule (26) enforces that for each E' in the points-to set of E for which the cast fails, the cast will still fail in P'.

[11] In [17], this constraint for cast-expressions reads $[E] \leq [(C)E]\,\mathbf{or}\,[(C)E] \leq E$. We can use a simplified version here because we make the assumption that the original program does not contain up-casts.

Interface-Compatibility Constraints Finally, we need to ensure that the customized program preserves interface-compatibility. To this end, we impose the additional constraint of rule (27) in Figure 4. This rules states that the types of declarations and expressions that occur in external class libraries cannot be changed.

4 Introducing Custom Classes

The introduction of custom classes proceeds using the following steps. First, the class hierarchy is extended with custom classes and auxiliary types. Then, type constraints are computed for the program with respect to this extended class hierarchy. This is followed by a step in which all constraints are transformed into simple equality or subtype constraints. Finally, the constraint system is solved in order to determine where custom classes can be used.

4.1 Extension of the Class Hierarchy

Before computing type constraints, we extend the class hierarchy with the custom classes that we would like to introduce. Adding these types *prior to* the construction of the constraints will allow us to determine where custom container classes *may be introduced.* In addition to the custom classes themselves, some auxiliary types will be added to the hierarchy. In what follows, we use the term *customizable class* to refer to a class for which we would like to introduce a custom version. Specifically, we extend the original class hierarchy as follows:

- For each customizable class C with superclass B, a class *CustomC* is created that contains methods and fields that are identical to those in C. If B is not customizable, then *CustomC*'s superclass is B, otherwise it is *CustomB*.
- For each customizable class C, a type $C^{\top}$ is introduced, and both C and *CustomC* are made a subtype of $C^{\top}$. Type $C^{\top}$ contains declarations of all methods in C that are not declared in any superclass of C.
- For each customizable container C, a type $C^{\perp}$ is introduced, and $C^{\perp}$ is made a subclass of both C and *CustomC*. Type $C^{\perp}$ contains no methods.

Multiple inheritance is used because it allows us to express that the type of an allocation site E should be either C or *CustomC* by way of subtype-constraints $[E] \leq C^{\top}$ and $C^{\perp} \leq [E]$. It is important to note that these multiple inheritance relations are *only* used during the solving of the type constraints and that the customized program does *not* refer to any type $C^{\top}$ or $C^{\perp}$.

Figure 6 shows the parts of the class hierarchy relevant for the customization of the example program of Figure 2 after adding the classes `CustomHashtable` and `CustomProperties`, and the additional types `Hashtable`$^{\top}$, `Hashtable`$^{\perp}$, `Properties`$^{\top}$ and `Properties`$^{\perp}$. Section 4.4 describes how the *CustomC* classes can be further transformed, and turned into a separate class hierarchy such as the one shown earlier in Figure 1.

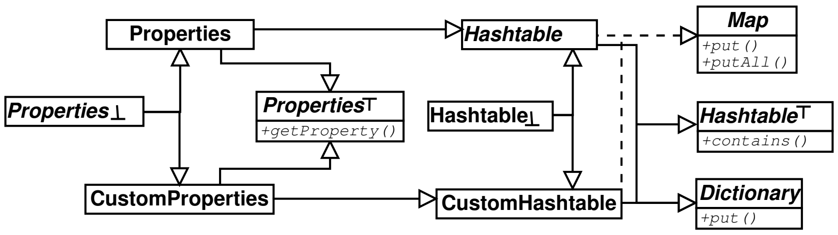

Fig. 6. Fragment of the collection hierarchy in `java.util.*` after extending it with custom classes and interfaces. For each class/interface, one or more of the methods defined/declared in that class/interface are shown.

The original program always allocates an object of type C at an allocation site `new` $C(E_1, \cdots, E_n)$, as was reflected by rule (22) in Figure 4. In the transformed program, we want to allow solutions where the allocated object is either of type C or of type *CustomC*. To this end, we replace rule (22) with rules (22)(a)–(22)(c) shown in Figure 7.

Figure 8 shows all non-trivial type constraints for the example program of Figure 2. For convenience, we have annotated each constraint in Figure 8 with the line number(s) in the source code from which it was derived, and with the number of the rule(s) in Figure 4 responsible for the constraint's creation.

4.2 Constraint Simplification

Constraints of the form $[E]{\leq}C_1 \textbf{or} \cdots \textbf{or} [E]{\leq}C_k$ (generated using rule (4)) give rise to bifurcations when traversing the solution space, and constraints of the form $[E']{\not\leq}[(C)E]$ (generated using rule (26)) make it impossible to define a monotone iteration step. Therefore, in order to simplify the process of constraint solving, we first perform a step in which all constraints are reduced to the simple forms $[x]{\leq}[y]$, $[x] = [y]$, and $[x]{\triangleq}[y]$.

Simplification of Disjunctions A virtual method call $E.m(E_1, \cdots, E_n)$ to a method M gives rise to a disjunction $[E]{\leq}C_1 \textbf{or} \cdots \textbf{or} [E]{\leq}C_k$ if $[E]$ has multiple supertypes $C_1, \cdots, C_k$ in which method $m(\cdots)$ is declared such that there is not a single method M that is overridden by all $C_i.m(\cdots)$, for all i, $1 \leq i \leq k$. Our approach will be to replace the entire disjunction by one of its branches $[E]{\leq}C_j$, for some j, $1 \leq j \leq k$. Note that by imposing a *stronger* constraint on $[E]$, we are potentially reducing the number of solutions of the constraint system. Nevertheless, at least one solution is guaranteed to exist: The original program fulfills each of the components of the original disjunction[12], so it will meet the simplified constraint as well.

Still, choosing the branch to replace the disjunction requires some consideration. Consider the constraint: [`h2`] $\leq$ `Map` **or** [`h2`] $\leq$ `Dictionary` that

[12] Note that the introduction of types $C^\top$ and $C^\perp$ does not affect this property, as they do not give rise to additional disjunctions.

program construct(s)/analysis facts	**implied type constraint(s)**	
expression $\texttt{new}\ \ C(E_1, \cdots, E_n)$ C is a customizable class	$[\texttt{new}\ C(E_1, \cdots, E_n)] \leq C^{\top}$ $C^{\perp} \leq [\texttt{new}\ C(E_1, \cdots, E_n)]$	(22)(a) (22)(b)
expression $\texttt{new}\ \ C(E_1, \cdots, E_n)$ C is a non-customizable class	$[\texttt{new}\ C(E_1, \cdots, E_n)] \triangleq C$	(22)(c)

Fig. 7. Revised type constraint rules for allocation sites and casts.

line	original constraint	rule	after simplification
3	$[\ \texttt{H1}\ ] \leq [\ \texttt{h1}\ ]$	(1)	
3	$[\ \texttt{H1}\ ] \leq \texttt{Hashtable}^{\top}$	(22)(a)	
3	$\texttt{Hashtable}^{\perp} \leq [\ \texttt{H1}\ ]$	(22)(b)	
4	$[\ \texttt{h1}\ ] \leq [\ \mathit{Param}(\texttt{JTree.JTree(),1})\ ]$	(8)	
4	$[\ \mathit{Param}(\texttt{JTree.JTree(),1})\ ]\ \texttt{=}\ \texttt{Hashtable}$	(27)	
5	$[\ \texttt{H2}\ ] \leq [\ \texttt{h2}\ ]$	(1)	
5	$[\ \texttt{H2}\ ] \leq \texttt{Hashtable}^{\top}$	(22)(a)	
5	$\texttt{Hashtable}^{\perp} \leq [\ \texttt{H2}\ ]$	(22)(b)	
6	$[\ \texttt{H3}\ ] \leq [\ \texttt{h3}\ ]$	(1)	
6	$[\ \texttt{H3}\ ] \leq \texttt{Hashtable}^{\top}$	(22)(a)	
6	$\texttt{Hashtable}^{\perp} \leq [\ \texttt{H3}\ ]$	(22)(b)	
7/15	$[\ \texttt{h3}\ ] \leq [\ \texttt{o}\ ]$	(10)	
8	$[\ \texttt{h3}\ ] \leq [\ \texttt{h2}\ ]$	(1)	
9	$[\ \texttt{h2}\ ] \leq \texttt{Map}$ **or** $[\ \texttt{h2}\ ] \leq \texttt{Dictionary}$	(4)	*(removed)*
10	$[\ \texttt{h2}\ ] \leq \texttt{Map}$	(4)	
11	$[\ \texttt{System.getProperties()}\ ] \leq [\ \texttt{p1}\ ]$	(1)	
11	$[\ \texttt{System.getProperties()}\ ]\ \texttt{=}\ \texttt{Properties}$	(27)	
12	$[\ \texttt{p1}\ ] \leq \texttt{Properties}$	(4)	
13	$[\ \mathrm{p1}\ ] \leq [\ \mathrm{h2}\ ]$	(1)	
16	$[\ \texttt{C1}\ ] \leq [\ \texttt{o}\ ]$	(12)	
16	$[\ \texttt{H3}\ ] \leq [\ \texttt{C1}\ ]$	(25)	
16	$[\ \texttt{S1}\ ] \not\leq [\ \texttt{C1}\ ]$	(26)	$[\ \texttt{C1}\ ] \leq \texttt{Hashtable}^{\top}$
16	$[\ \texttt{C1}\ ] \leq [\ \texttt{h4}\ ]$	(1)	
17	$[\ \texttt{h4}\ ] \leq \texttt{Hashtable}^{\top}$	(4)	
20	$\texttt{S1} \leq [\ \texttt{s}\ ]$	(1)	
20	$[\ \texttt{S1}\ ]\ \texttt{=}\ \texttt{String}$	(27)	
21/15	$[\ \texttt{s}\ ] \leq [\ \texttt{o}\ ]$	(10)	

Fig. 8. Type constraints for the example program of Figure 2(a) as derived according to Figure 4. The rows in the table show, from left to right, the line in the source code from which the constraint was derived, the constraint itself, the rule that triggered the creation of the constraint, and the constraint after simplification (where applicable).

was generated due to the call `h2.put("FOO", "BAR")` in the example program of Figure 2. If we simplify this constraint to: [`h2`] $\leq$ `Dictionary`, we obtain a constraint system in which variable `h2` must be a subtype of both `Map` and `Dictionary`, as well as a supertype of `java.util.Properties`. This implies that `h2`'s type must be a subtype `java.util.Hashtable`, which, in turn, requires that allocation sites `H2` and `H3` must remain of type `java.util.Hashtable`, preventing us from customizing these allocation sites. On the other hand, replacing the original disjunction with: [`h2`] $\leq$ `Map` allows us to infer the solution shown earlier in Figure 2(b), in which allocation sites `H2` and `H3` have been customized. Clearly, some choices for simplifying disjunctions are better than others.

We use the following approach for the simplification of disjunctions. First, any constraint $[x]{\leq}C_1 \textbf{or} \cdots \textbf{or} [x]{\leq}C_n$ for which there already exists another constraint $[x]{\leq}C_j$ can simply be removed by subsumption, as the latter constraint implies the former. Second, we use the heuristic that any constraint $[x]{\leq}C_1 \textbf{or} \cdots \textbf{or} [x]{\leq}C_n$, for which there exists another constraint $[y]{\leq}C_j$, for some unique j $(1 \leq j \leq n)$ such that $\mathit{PointsTo}(P, x) \cap \mathit{PointsTo}(P, y) \neq \emptyset$. is simplified to $[x]{\leq}C_j$. If no constraint $[y]{\leq}C_j$ exists, the disjunction is simplified by making an arbitrary choice. The results of this approach have been satisfactory so far. If the loss of precision becomes a problem, one could compute the results obtained for all possible choices for each disjunction, and select a maximal solution.

Simplification of $\not\leq$-Constraints Constraints of the form $[E']{\not\leq}[(C)E]$ are introduced by rule (25) in order to preserve the behavior of casts that may fail. For example, in the program of Figure 2 the cast on line 14 fails when method `bar()` is called from method `bad()`, because in this case `o` will point to a `String`-object that was allocated at allocation site `S1`.

Our approach will be to introduce additional constraints that are sufficient to imply the $\not\leq$-constraint. Specifically, for each cast $(C)E$ for which the points-to set $\mathit{PointsTo}(P, E)$ contains an expression E' such that $[E']_P{\not\leq}C$, we introduce a constraint $[(C)E]{\leq}\ C^{\top}$. This additional constraint prevents the generalization of the target type of a cast in situations where that would change a failing cast into a succeeding cast.

It is easy to see that the addition of this constraint ensures that the behavior of failing casts is preserved in the customized program P'. Suppose that $(C)E$ is a cast that may fail in the original program P. Then, there exists an $E' \in \mathit{PointsTo}(P, E)$ for which $[E']_P{\not\leq}C$. Since P does not instantiate any custom classes, we also know that $[E']_P{\not\leq}\mathit{CustomC}$, and therefore that $[E']_P{\not\leq}C^{\top}$. Hence, requiring that $[(C)E]{\leq}C^{\top}$ ensures that the constraint $[E']{\not\leq}[(C)E]$ is satisfied in P'.

Example. Figure 8 shows the simplified type constraints for the program of Figure 2. For the disjunction [`h2`] $\leq$ `Map` **or** [`h2`] $\leq$ `Dictionary` in Figure 8, there already exists another, stronger constraint [`h2`] $\leq$ `Map`, so it can simply be removed. Furthermore, the $\not\leq$-constraint [`S1`] $\not\leq$ [`C1`] is replaced with a constraint [`C1`] $\leq$ `Hashtable`$^{\top}$.

4.3 Solving the Constraints

Now that all constraints are of the forms $E \leq E'$, $E = E'$, and $E \triangleq E'$ solving the constraint system is straightforward. First, we create a set of equivalence classes of declaration elements and expressions that must have exactly the same type, and we extend the $\leq$ relationship to equivalence classes in the obvious manner. Then, we compute the set of possible types for each equivalence class using an optimistic algorithm. This algorithm associates a set S_E of types with each equivalence class E, which is initialized as follows:

– For each equivalence class E that contains an allocation expression $E \equiv$ `new` C, S_E is initialized to contain the types C and *CustomC*.
– For each equivalence class E that does not contain any allocation expressions, S_E is initialized to contain all types except $C^{\top}$ and $C^{\perp}$, for all C.

Then, in the iterative phase of the algorithm, the following steps are performed repeatedly until a fixpoint is reached:

– For each pair of equivalence classes D, E such that there exists a type constraint $D \leq E$, we remove from S_D any type that is not a subtype of a type that occurs in S_E.
– For each pair of equivalence classes D, E such that there exists a type constraint $D \leq E$, we remove from S_E any type that is not a supertype of a type that occurs in S_E.

Termination of this algorithm is ensured because each iteration decreases the number of elements in at least one set, and there is a finite number of sets. Moreover, each equivalence class will contain at least the type that is associated with its elements in the original program.

equivalence class	**possible types**
{ `p1, Properties` }	{ `Properties` }
{ `h1` }	{ `Hashtable` }
{ `H1` }	{ `Hashtable` }
{ `h2` }	{ `Map, Hashtable, CustomHashtable` }
{ `H2` }	{ `Hashtable, CustomHashtable` }
{ `h3` }	{ `Map, Hashtable, CustomHashtable` }
{ `H3` }	{ `Hashtable, CustomHashtable` }
{ `h4` }	{ `Hashtable, CustomHashtable` }
{ `C1` }	{ `Hashtable, CustomHashtable` }
{ `o` }	{ `Object` }
{ `s` }	{ `String` }

Fig. 9. Possible types computed for each equivalence class.

Figure 9 shows the sets of types computed for each of the equivalence classes in our example. The interpretation of these sets of types requires some remarks:

– Figure 9 depicts many possible solutions. In each solution, a single type in S_E is chosen for each equivalence class E.
– If type T occurs in S_E, then at least one solution to the constraint system exists in which the elements in E have type T.
– Selecting types for different equivalence classes can in general not be done independently. For any given pair of equivalence classes D and E, choosing an arbitrary element in S_D for equivalence class D, and an arbitrary element in S_E for equivalence class E may result in a type-incorrect program.
– The previous observation particularly applies to two equivalence classes associated with allocation sites A_1 and A_2. Selecting type C for (the equivalence class containing) A_1 may prevent us from selecting type *CustomC* for (the equivalence class containing) A_2. For example, if a call `bar(h2)` is added to method `Example.main()`, we have the choice of: (i) customizing both `H2` and `H3` or (ii) not customizing both `H2` and `H3`. However, customizing `H2` but not `H3` (or vice versa) will not preserve the behavior of cast `C1`.
– However, we conjecture that a solution exists in which type *CustomC* is selected for *all* equivalence classes E such that $CustomC \in S_E$.

A more precise treatment of these properties is currently in progress.

4.4 Pragmatic Issues and Further Customization

There are several issues that require straightforward extensions to our basic approach. These include the treatment of subtypes of standard library classes (e.g., an application declaring a class `MyHashtable` that extends `Hashtable`), and limiting the introduction of custom classes in the presence of serialization. Space limitations prevent us from providing more details.

Thus far, we have presented how variables and allocation sites of type C can be updated to refer to type *CustomC*. At this point it has become easy to replace *CustomC* with a small hierarchy of custom classes such as the one shown in Figure 1 by applying refactorings [5, 17] as follows:

– Split class *CustomC* into an abstract superclass *AbstractCustomC* (that only contains abstract methods) and a concrete subclass *CustomC*. Declarations and casts (but not allocation sites) that refer to type *CustomC* are made to refer to *AbstractCustomC* instead.
– At this point, clones $CustomC_1, \cdots, CustomC_n$ of class *CustomC* can be introduced as a subclass of *AbstractCustomC*. Any allocation site of type *CustomC* may be updated to refer to any $CustomC_i$.

The next section will discuss a number of optimizations that can be (independently) applied to each $CustomC_i$.

5 Implementation

We use the *Gnosis* framework for interprocedural context-sensitive analysis developed at IBM Research to compute all static analysis information that is needed for customization. Two nonstandard components of our analysis are:

- For customizable classes, each allocation site in user code is analyzed separately, but a single logical site represents all allocations in system code.
- Analysis is done in two passes: a conventional points-to analysis [14, 8, 15] is followed by a step in which additional data flow facts are introduced that model the type constraints due to method overriding, similar in spirit to [7].

Like many static whole-program analysis and transformation tools (e.g., [18]), *Gnosis* relies on the user to specify the behaviors of native library methods as well as any uses of reflection in order to compute a safe analysis result.

In order to gather profile information, the customization framework itself is used to replace the standard library classes created by an application with custom versions that gather profile information. This is subject to the usual limitations. In other words, we cannot gather profile information for container objects in cases where interface-compatibility constraints prevent us from applying customization. Usage statistics are gathered per allocation site and include:

1. A distribution of the construction-time sizes.
2. A distribution of the high-watermarks of the sizes of the container objects allocated at the site (i.e., the largest size of containers during their life-time).
3. Distributions of the container's size at method invocations (per method).
4. The hit-rates of search operations.
5. The hit-rates of several caching schemes for optimizing search operations.

Distributions (1) and (2) are used as a basis for deciding on an initial allocation size and on lazy vs. eager allocation. Combined with (3), they are also used to determine whether providing special treatment for singleton containers is beneficial. Distribution (3) is also used to determine whether or not we want to optimize certain methods for specific sizes, such as empty or singleton containers. Distribution (4) is used to decide on whether or not search methods are to be optimized for succeeding or failing searches. Finally, (5) is used to decide on caching schemes. In our implementation, optimization decisions are based on thresholds, such as hit-rates for search operations, cache schemes, fractions of containers that remain empty or singletons, etc. All thresholds used are very high. For example, we use lazy allocation if 75% of the allocated containers remain empty or contain only one item. Similarly, caching schemes are used only if their hit-rates are 90% or higher.

The optimizations that are incorporated into custom classes include:

1. Caching the last retrieved items in a container using different caching schemes.
2. Lazy allocation of encapsulated data structures such as a `Hashtable`'s array of hash-buckets,
3. Selecting a non-default initial size and growth-strategy for a container's underlying data structures, depending on the success-rate of retrieval operations, the distribution of the high-watermarks of the container sizes, etc.
4. Efficiently implementing frequently occurring corner cases such as container classes that often contain zero or one elements. For example, it is often possible to use a single, shared `EmptyIterator` whose `hasNext()` method always returns false.

5. Transforming instance fields into class fields if their values are identical for all objects allocated at some allocation site, or if the differences are non-critical.
6. Specialization of container classes for the type of objects stored in them, if static analysis can determine these types. Examples of such optimizations are `Integer` keys in `Hashtable`s, for which we can store the `int` values instead, or `String`s for which can exploit the fact that their hashcodes are cached.
7. Finalizing classes that have no subtypes in our program.
8. Removal of unnecessary synchronization. Currently, we only remove synchronization if an application is single-threaded. This is the case if the `Thread.start()` method is not reachable in the call-graph starting at the program entry point[13].

In addition, static analysis information is used to detect situations where certain methods are never invoked on a container object that originates from a given allocation site A. This information is used to remove methods and fields from the custom class used at A. The bookkeeping fields used for implementing fail-fast iterators are an example of a situation where this is useful.

Java bytecode [10] is generated for each custom class by preprocessing a template implementation of a library class, and compiling the resulting source file to Java .class files. In the current implementation, this is done using the standard C-preprocessor. JikesBT[14], a byte-code instrumentation tool developed at IBM Research, is used for the rewriting of the application's class files so that they refer to the generated custom classes.

6 Experimental evaluation

To evaluate our techniques, we measured the execution times and memory footprint of a number of Java applications on a workstation (hyperthreaded Pentium 4 at 2.8 GHz, 1GB RAM) running Linux 2.4.21 and two Java virtual machines: IBM's "j9" VM that is being distributed with IBM's WebSphere Studio Device Developer product and Sun's Hotspot Server 1.3.1 JVM. All measurements were performed using a maximum heap of 400 MB.

Three of our benchmark programs are taken from the SPECjvm98 suite: _202_jess is an expert shell system, _209_db is a memory resident database, and _218_jack is a parser generator. The other benchmarks we include are HyperJ (an aspect-oriented development tool), Jax [18], PmD (a open-source tool available from SourceForge for detecting programming errors) and a chess program developed at IBM. All execution times were measured using a harness: each program is executed 10 times within one invocation of the VM, and we report the fastest time of the 10 runs. With the exception of HyperJ, for which we had only one input data set available, all measurements were performed using larger data sets than the training sets used to collect profile information.

[13] Here, "Program entry point" refers to the entry point of the actual benchmark program, and not of the harness used for measuring execution time.
[14] See `www.alphaworks.ibm.com/tech/jikesbt`

	Sun HotSpot(TM) Server 1.3.1				IBM J9 2.2			
	orig	cust1	cust2	cust3	orig	cust1	cust2	cust3
_202_jess	1.92s	1.83s *(1.05)*	1.42s *(1.36)*	1.41s *(1.37)*	1.38s	1.45s *(0.95)*	1.21s *(1.14)*	1.21s *(1.15)*
_209_db	15.04s	12.23s *(1.23)*	8.45s *(1.78)*	8.51s *(1.77)*	8.99s	8.63s *(1.04)*	6.48s *(1.39)*	6.48s *(1.39)*
_218_jack	3.23s	3.18s *(1.02)*	2.28s *(1.42)*	2.24s *(1.44)*	1.88s	1.93s *(0.97)*	1.51s *(1.24)*	1.40s *(1.34)*
Jax	21.46s	21.24s *(1.01)*	20.97s *(1.02)*	19.93s *(1.08)*	16.36s	15.04s *(1.09)*	15.00s *(1.09)*	14.90s *(1.10)*
HyperJ	13.03s	12.77s *(1.02)*	11.04s *(1.18)*	10.83s *(1.20)*	10.04s	9.63s *(1.04)*	8.68s *(1.16)*	8.58s *(1.17)*
Chess	18.07s	18.23s *(0.99)*	18.23s *(0.99)**	18.23s *(0.99)**	6.93s	6.59s *(1.05)*	6.59s *(1.05)**	6.59s *(1.05)**
Pmd	5.43s	5.33s *(1.02)*	5.33s *(1.02)**	5.33s *(1.02)**	4.37s	3.76s *(1.16)*	3.76s *(1.16)**	3.76s *(1.16)**
GEOMEAN		***(1.05)***	***(1.23)***	***(1.24)***		***(1.05)***	***(1.17)***	***(1.19)***

Fig. 10. Execution times and speedups obtained through customization. The execution times presented are of the original programs (`orig`), the programs with customized container classes, but without synchronization elimination(`cust1`), the programs with customized containers classes of which we eliminated the synchronization where possible (`cust2`), and the programs with customized and desynchronized containers classes and desynchronized `StringBuffer`s (`cust3`).

	zipped archive		heap size	
	orig	cust3	orig	cust3
_202_jess	173KB	175KB *(1.01)*	1.74MB	1.77MB *(0.99)*
_209_db	6KB	21KB *(3.68)*	9.57MB	9.57MB *(1.00)*
_218_jack	70KB	95KB *(1.35)*	15.78MB	8.54MB *(0.54)*
Jax	582KB	615KB *(1.06)*	44.23MB	40.98MB *(0.93)*
HyperJ	1,767KB	1,821KB *(1.03)*	44.42MB	41.38MB *(0.93)*
Chess	135KB	151KB *(1.12)*	9.33MB	9.29MB *(1.00)*
Pmd	498KB	525KB *(1.05)*	142.51MB	123.72MB *(0.87)*
GEOMEAN		***(1.30)***		***(0.88)***

Fig. 11. Archive size increase and memory footprint reduction resulting from customization.

In Figure 10 we report execution times of four versions of the benchmarks, that were customized in the exact same way for both VMs.[15] The consequences of the customizations on memory consumption are depicted in Figure 11. An analysis of the obtained results and applied customizations reveals that:

– In _202_jess, the keys used in hashtables are either Strings or Integers, and on 2 of the hashtables all search operations fail. Depending on the VM used, customizing the Hashtable class for this usage pattern resulted in a 5% speedup or in a 5% slowdown. Eliminating synchronization, including the synchronization on very frequently used Vector objects, results in speedups between 15% and 37%.

– In _209_db, 99% of all consecutive retrieval operations on Vectors retrieve the same element (during what is essentialy a column-major-order operation on

[15] In order to evaluate the customizations correctly, the original programs need to be executed with the original library classes before customization. Since all our customized classes are customized versions of IBM's jclMax implementation (which is distributed with IBM's WebSphere Studio Device Developer product), we enforced the use of the original jclMax classes on both VMs by prepending them to the boot classpath of the VMs.

a row-major-order stored array of Vectors), and the application of a caching scheme results in a 23% speedup on the Sun VM. Additionally removing the synchronization on these Vectors results in a 77% speedup on the Sun VM, and a speedup of 39% on IBMs j9.
– In _228_jack, 99% of all search operations are on empty hashtables (of which a lot are allocated), or hashtables containing one element only. Using lazy allocation and eliminating the bookkeeping data for fail-safe iteration, we can reduce the heap memory consumption with 46%. On Sun's VM, the resulting overhead in the retrieval operations can be compensated by optimizing the retrieval for the corner case of hashtables containing one element only. On IBM's j9, the overhead is almost compensated. Finally, removing synchronization results in speedups between 34 and 44%.
– In HyperJ, the same situation occurs, and lazy allocation and the elimination of unncessary bookkeeping data for the hot allocation sites results in speedups around 2-4% if no synchronization is eliminated, and 17-20% if synchronization is eliminated. Memory consumption drops with 7%. Especially for this benchmark, we should note that our customization results are obtained on top of the already applied manual fine-tuning by the programmers via construction time parameterization.
– In Jax, most containers remain very small, and adapting the initial container size to reflect that results in speedups ranging between 1 and 3%. Memory consumption as a result drops with 7%.
– In PmD, the vast majority of a huge number of allocated HashMaps remains empty or contains only one element. Lazy allocation, the elimination of bookkeeping data for fail-safe iteration and the optimization of access methods result in speedups of 2 to 16%, and a reduction of the allocated memory with 13%. Since PmD contains a multi-threaded GUI front-end, no synchronization was removed.
– In the chess program, enumerations (over piece positions) stored in hashtables occur very frequently. Optimizing the number of hash-buckets for the number of positions (which seemed limited to the number of pieces on a board) resulted in speedups between 2 and 16%, depending on the VM. Like Pmd, the chess program contains a multi-threaded front-end. As a result, we did not yet try to eliminate any synchronization.

On average, the customizations excluding the elimination of synchronization result in speedups of 5% on both VMs. By additionally eliminating synchronization on container classes and StringBuffers in the single-threaded programs, an average speedup of 19-24% can be obtained. The elimination of synchronization is therefore clearly the dominant optimization, in particular the elimination of synchronization on container classes. There are exceptions to this trend however: for Jax the speedup following synchronization removal is insignificant on j9, and for _218_jack the removal of synchronization on StringBuffers is much more significant (10% additional speedup) than for the other programs.

The raw execution times shown in Figure 10 indicate that the two VMs have somewhat different performance characteristics. It is therefore no surprise that the obtained speedups are different for each VM as well. This indicates that the decision logic used for the customization should be made parameterizable for specific VMs.

Finally, we should note that program archive size increases only by a small amount because of the customization: as indicated in Figure 11 at most 54KB is added to the archives, for HyperJ. For all but the smallest programs the zipped archives grow by 12% or less.

7 Related Work

Yellin [19] and Högstedt et al. [9] discuss techniques for automatically selecting optimal component implementations. In [19], selection takes place at run-time, and based on on-line profiling only, while in [9] off-line profiling is used as well. In both cases, the component developer is required to provide all component versions up front, making them less viable when many orthogonal implementation decisions exist, as is the case in the present paper. Unlike our work, the approaches of [19, 9] do not require static analysis because programs are correct by construction. In our setting, static analysis is needed to guarantee type-correctness in cases where objects are exchanged with the standard libraries (or other components). However, as the approaches of [19, 9] do not rely on static analysis, they are incapable of *eliminating* functionality from classes.

Schonberg et al. [16] discuss techniques for automatically selecting concrete datastructures to implement the abstract datatypes set and map in SETL programs. Depending on whether or not iterators and set-theoretic operations such as union and intersection are applied to abstract datatypes, their optimizing compiler selects an implementation from a predetermined collection of implementations in which sets are represented as linked lists, hashtables, or bit-vectors.

Transformational programming [2] is a programming methodology based on top-down stepwise program refinement. Here, the programmer specifies a program in terms of very high-level constructs without worrying about efficiency. This program is made more efficient by automatically applying a sequence of finite difference transformations or by applying incremental refinement [4, 13].

There is a large body of work on automatic optimization of data structures in specific domains (e.g., linear algebra kernels). For example, the Berkeley Benchmarking and Optimization Group (see `http://bebop.cs.berkeley.edu`) studies issues related to optimization and data structure selection for sparse matrix multiplication problems. In the same domain, Yotov et al. [20] compare empirical and model-driven approaches for selecting customized data structures.

There is also work on optimizations that can be applied to specific containers such as hashtables. Beckmann and Wang [1] discuss the optimization of hashtables by prefetching the objects that are most likely to be searched for, and Friedman et al. [6] discuss the optimization of the maximal access time of hashtables to improve real-time behavior by, e.g., incrementalizing rehash operations.

The elimination of synchronization has been a very popular research subject (see for example [3]). To the best of our knowledge, all currently known solutions are either based on manual rewriting of a program to remove the synchronization or on bytecode annotations that can only be read by adapted VMs. Our approach of automatically replacing the allocation of standard types with the allocation of customized types allows to automate the existing techniques for synchronization elimination without requiring special VM support.

Type constraints were introduced [12] as a means to check whether a program conforms to a language's type system. If a program satisfies all type constraints, no type violations will occur at run-time (e.g., no method $m(\cdots)$ is invoked on an object whose class does not define or inherit $m(\cdots)$). Type constraints were recently used to check the preconditions and determine the allowable source-code modifications associated with generalization-related refactorings [17]. The problem of determining where custom versions of library classes may be introduced is very similar to the problem of determining declarations that can be updated by the EXTRACT INTERFACE refactoring. However, in [17] the type of a declaration is only replaced with one of its supertypes, whereas our work is unique in the sense that custom classes appear in a different branch of the class hierarchy.

8 Conclusions and Future Work

We have presented an automated approach for customizing Java container classes that relies on static analysis for determining where custom container classes may be introduced in an application, and on profile information for determining what optimizations are likely to be profitable. The approach was evaluated on a set of benchmark applications, and we measured speedups of up to 78%, averaging at 19-24%. The memory footprint reductions we measured range from -1 to 46%, averaging at 12%. The cost of the applied customizations in terms of code size is limited to 12% for all of the smallest programs we evaluated.

We plan to develop a more precise formal treatment of the properties of our algorithm for determining where custom allocation sites may be introduced. Other topics for future work include more advanced program transformations (e.g, replacing a `Hashtable` with an extra field in each key object that stores a reference to its corresponding value), applications to other library classes and the use of escape analysis to determine where unnecessary synchronizations can be removed from multi-threaded programs.

References

1. BECKMANN, B., AND WANG, X. Adaptive prefetching Java objects. manuscript.
2. CAI, J., AND PAIGE, R. Towards increased productivity of algorithm implementation. In *Proc. 1st ACM SIGSOFT symposium on Foundations of software engineering* (1993), pp. 71–78.
3. CHOI, J.-D., GUPTA, M., SERRANO, M. J., SREEDHAR, V. C., AND MIDKIFF, S. P. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Transactions on Programming Languages and Systems 25*, 6 (2003), 876–910.

4. Dewar, R. K., Arthur, Liu, S.-C., Schwartz, J. T., and Schonberg, E. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems (TOPLAS) 1*, 1 (1979), 27–49.
5. Fowler, M. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.
6. Friedman, S., Leidenfrost, N., Brodie, B., and Cytron, R. Hashtables for embedded and real-time systems. In *IEEE Real-Time Embedded System Workshop* (2001).
7. Glew, N., and Palsberg, J. Type-safe method inlining. In *Proc. 16th European Conference on Object-Oriented Programming* (2002), pp. 525–544.
8. Hind, M., and Pioli, A. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming 39*, 1 (2001), 31–55.
9. Högstedt, K., D., K., Rajan, V., Roth, T., Sreedhar, V., Wegman, M., and Wang, N. The autonomic performance prescription. Available from the author at `wegman@watson.ibm.com`.
10. Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.
11. Palsberg, J. Type-based analysis and applications. In *Proc. ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)* (Snowbird, UT, 2001), pp. 20–27.
12. Palsberg, J., and Schwartzbach, M. *Object-Oriented Type Systems.* John Wiley & Sons, 1993.
13. Pavlovic, D., and Smith, D. Software development by refinement. In *UNU/IIST 10th Anniversary Colloqium, Formal Methods at the Crossroads: From Panaea to Foundational Support.* Springer-Verlag, 2003.
14. Rountev, A., Milanova, A., and Ryder, B. Points-to analysis for Java using annotated constraints. In *Proc. 16th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)* (Tampa Bay, FL, 2001), pp. 43–55.
15. Ryder, B. G. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. 12th International Conference on Compiler Construction (CC 2003)* (Warsaw, Poland, April 2003), pp. 126–137.
16. Schonberg, E., Schwartz, J., and Sharir, M. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems 3*, 2 (April 1981), 126–143.
17. Tip, F., Kieżun, A., and Bäumer, D. Refactoring for generalizations using type constraints. In *Proc. 18th Annual Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)* (2003).
18. Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., and Streeter, D. Practical extraction techniques for Java. *ACM Transactions on Programming Languages and Systems 24*, 6 (2002), 625–666.
19. Yellin, D. Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal 42*, 1 (January 2003), 85–97.
20. Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., and Wu, P. A comparison of empirical and model-driven optimization. In *Proc. ACM SIGPLAN 2003 conference on Programming language design and implementation* (2003), pp. 63–76.