Compact and Efficient Strings for Java

Christian Häubl*, Christian Wimmer, Hanspeter Mössenböck

Institute for System Software, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University Linz, Austria

Abstract

In several Java VMs, strings consist of two separate objects: metadata such as the string length are stored in the actual string object, while the string characters are stored in a character array. This separation causes an unnecessary overhead. Each string method must access both objects, which leads to a bad cache behavior and reduces the execution speed.

We propose to merge the character array with the string's metadata object at run time. This results in a new layout of strings with better cache performance, fewer field accesses, and less memory overhead. We implemented this optimization for Sun Microsystems' Java HotSpot™ VM, so that the optimization is performed automatically at run time and requires no actions on the part of the programmer. The original class `String` is transformed into the optimized version and the bytecodes of all methods that allocate string objects are rewritten. All these transformations are performed by the Java HotSpot™ VM when a class is loaded. Therefore, the time overhead of the transformations is negligible.

Benchmarks show an improved performance as well as a reduction of the memory usage. The performance of the SPECjbb2005 benchmark increases by 8%, and the average used memory after a full garbage collection is reduced by 19%. The peak performance of SPECjvm98 is improved by 8% on average, with a maximum speedup of 62%.

Key words: Java, String, Optimization, Performance

1. Introduction

Strings are one of the essential data structures used in nearly all programs. Therefore, string optimizations have a large positive effect on many applications. Java supports string handling at the language level [1]. However, all string operations are compiled to normal method calls of the classes `String` and `StringBuilder` in the Java bytecodes [2].

To the best of our knowledge, strings in Sun Microsystem's Java HotSpot™ VM, Oracle's JRockit, and IBM's J9 use the object layout illustrated in Figure 1 (a). Every string is composed of two objects: metadata such as the string length are stored in the actual string object, whereas the string characters are stored in a separate character array. This allows several string objects to share the same character array. To increase the opportunities for sharing the character array, string objects use the fields `offset` and `count`. These fields store the string's starting position within the character array and the string length, so that a string does not need to use the full character array. This is beneficial for methods such as `String.substring()`: a new string object that references the same character array is allocated, and only the string's starting position and length are set accordingly. No characters must be copied.

If a string uses its whole character array, the field `count` is a duplication of the character array's field `length`. Furthermore, the field `offset` is an overhead that reduces the performance: when a string character is accessed, `offset` is loaded to determine the start of the string within the array.

*Corresponding author

Email addresses: `haeubl@ssw.jku.at` (Christian Häubl), `wimmer@ssw.jku.at` (Christian Wimmer), `moessenboeck@ssw.jku.at` (Hanspeter Mössenböck)

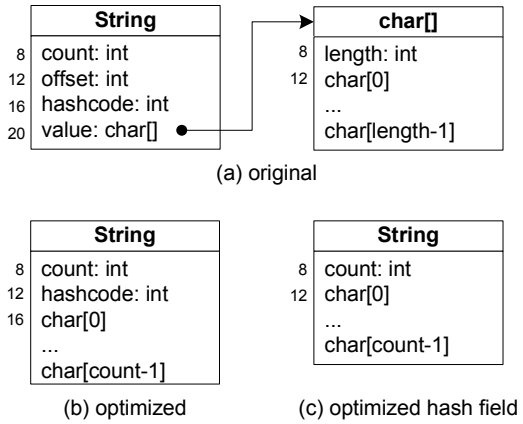

Figure 1: Object layout of strings

Although string objects can share their character arrays, this is not the common case. We measured the percentage of strings that do not use their full character array: 0.05% for the SPECjbb2005 [3] benchmark, 5% for the SPECjvm98 [4] benchmarks, 14% for the DaCapo [5] benchmarks, and 29% for the SPECjvm2008 [6] benchmarks. However, a string also shares its character array when it is explicitly copied using the constructor `String(String original)`. Of all string allocations, 19% are explicit string copies for the SPECjbb2005 benchmark and 4% for the SPECjvm98 benchmark. The DaCapo and the SPECjvm2008 benchmarks hardly allocate any explicit string copies.

Because of these results, we propose different string layouts as shown in Figure 1 (b) and (c). In the first variant, we remove the field `offset` and merge the character array with the string object, which also makes the fields `count` and `value` unnecessary. This precludes the sharing of character arrays between string objects, but has several advantages such as the reduction of memory usage and the elimination of field accesses. The second variant, described in Section 4.6, also removes the field `hashcode` to save another four bytes per string object. The computed hash code is cached in the object header instead.

This paper is an extended version of an earlier conference paper [7]. It contributes the following:

- We present two string optimization variants that reduce the memory usage and increase the performance. Our approach requires neither actions on the part of the programmer nor any changes outside the Java VM.
- We present details of the integration into both the client and the server compiler of the Java HotSpot™ VM.
- We discuss compatibility issues of our optimization with Java-specific features such as the Java Native Interface (JNI) and reflection. Although our prototype implementation does not yet fulfill the Java specification completely, it is capable of executing all Java applications and benchmarks we tried.
- We evaluate the impact of our optimization on the number of allocated bytes and the performance of the SPECjbb2005, SPECjvm98, SPECjvm2008, and DaCapo benchmarks.

The paper is organized as follows: Section 2 gives a short overview of the relevant subsystems in the Java HotSpot™ VM and illustrates where changes were necessary. Section 3 discusses the advantages of our optimization. Section 4 describes the key parts of our implementation, i.e., bytecode transformation and string allocation. Section 5 presents the benchmark results. Section 6 deals with related work, and Section 7 concludes the paper.

2. System Overview

We build on an early access version of Sun Microsystems' Java HotSpot™ VM that is part of the upcoming JDK 7 [8]. The VM is available for multiple architectures, however our string optimization is currently only implemented for the IA-32 architecture because platform-dependent code is necessary within the interpreter and the just-in-time (JIT) compilers.

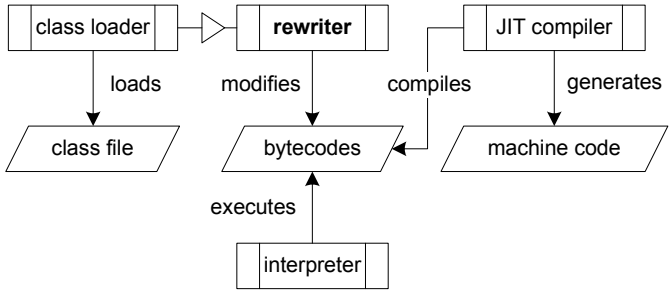

Figure 2: System overview

Figure 2 illustrates some of the subsystems necessary for the execution of bytecodes. When a class is loaded by the class loader, the corresponding class file is parsed and verified, run-time data structures such as the constant pool and the method objects are built, and finally the interpreter starts executing the bytecodes. For every method, the number of invocations is counted in order to detect so-called hotspots. When an invocation counter exceeds a certain threshold, the JIT compiler compiles the method's bytecodes to optimized machine code. There are two different JIT compilers for the Java HotSpot™ VM:

- The *client compiler* is optimized for compilation speed and refrains from time consuming optimizations [9, 10]. With this strategy, the application startup time is low while the generated machine code is still reasonably well optimized. The compilation of a method is performed in three phases: high-level intermediate representation (HIR) generation, low-level intermediate representation (LIR) generation, and code generation.

  The HIR represents the control flow graph. Since Java 6, it is in static single assignment (SSA) form [11] and suitable for global optimizations. For generating the HIR, two passes over the bytecodes are necessary. The first pass determines and creates the basic blocks for the control flow graph. The second pass uses abstract interpretation of the bytecodes to link the basic blocks and to fill them with instructions. Several optimizations like constant folding, null-check elimination, and method inlining are performed during and after the generation of the HIR. Because the HIR is in SSA form, these optimizations are fast and can be applied easily.

  In the next phase, the LIR is created from the optimized HIR. The LIR is more suitable for register allocation and code generation because it is similar to machine code. However, the LIR is still mainly platform independent. Instead of physical machine registers, virtual registers are used for nearly all instructions. Platform-dependent parts, like the usage of specific machine registers for some instructions, can be modeled directly in the LIR. This simplifies the code generation. Linear scan register allocation [12] is used to map all virtual registers to physical ones.

  Code generation finishes the compilation of a method by translating each LIR instruction to the platform-dependent instructions. Uncommon cases like throwing an exception or invoking the garbage collector are handled as separate cases and are emitted at the method's end. In addition to the machine code, other meta data required for the execution is generated.

- The *server compiler* makes use of more sophisticated optimizations to produce better code [13]. It is designed for long-running server applications where the initial compilation time is irrelevant, and where a high peak performance is essential. Like a traditional compiler, the server compiler uses the

following phases for compilation: parsing, platform-independent optimization, instruction selection, global code motion and scheduling, register allocation, peephole optimization, and code generation.

The parser creates the compiler's intermediate representation (IR) from the bytecodes and applies some optimizations like constant folding. Platform-independent optimizations like null-check elimination and dead-code removal are applied to the IR. The instruction selection phase maps the platform-independent IR to platform-dependent instructions. These are reordered to optimize their sequence. This avoids dependencies between the instructions and increases the performance. Then, physical machine registers are assigned to each instruction. A peephole optimization analyzes the code in small pieces and merges or replaces instructions. The last step generates the machine code and additional information required for the execution.

Within the Java HotSpot™ VM, certain methods can be declared as intrinsic. When such a method is compiled or inlined, a handcrafted piece of machine code is used as the compilation result. This allows optimizing specific methods manually.

Every Java object has a header of two machine words, i.e., 8 bytes on 32-bit architectures and 16 bytes on 64-bit architectures [14]. Figure 3 shows this object layout for a character array. The first machine word, the so called mark word, stores the identity hash code and the object's synchronization status. The identity hash code is a random value that does not depend on the object's content, and is calculated via the method `System.identityHashCode()`. Caching is necessary because its value must not change during the object's lifetime. On 32-bit architectures, the identity hash code is truncated to 25 bits. On 64-bit architectures, the mark word is large enough to hold the identity hash code without truncation. The identity hash code is preferably unique for each object but it is not guaranteed to be so.

The second header word stores a pointer to the class descriptor object that is allocated when a class is loaded. The class descriptor holds the metadata and the method table of the class. Then, the object data follows, i.e., the array length and the characters in case of a character array.

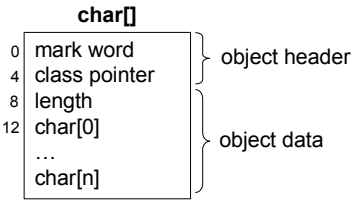

Figure 3: Java object header on 32-bit architectures

As shown in Figure 2, we add a *rewriter* component to the basic execution system. After the class loader has finished loading a class, the *rewriter* checks if a method allocates string objects. If so, the method bytecodes are transformed. This is done only once per class and adds a negligible overhead to the execution time. Additionally, the string class itself is transformed manually. Several other subsystems of the VM are affected by our string optimization because `String` is a well-known class that is directly used within the VM. Nevertheless, we tried to minimize the number of changes.

3. Advantages of the Optimization

Although character array sharing between multiple string objects is no longer possible for optimized strings, the optimization has several other advantages:

- *Elimination of field accesses*: By removing the fields `offset`, `count`, and `value`, field access are saved in almost every string operation.
- *Reduced memory usage*: Original string objects have a minimum size of 36 bytes. With our optimization, the minimum size is 12 or 16 bytes, depending on the removal of the field `hashcode`. Saving up

to 24 bytes per string object results in the reduced memory usage shown in Figure 4, which depends on the string length. For example, for strings of length 10 our optimized string handling saves 37% of string memory, or 44% if also the field `hashcode` is eliminated. The figure also contains the average string lengths for the different benchmark suites. Because the memory usage is reduced, fewer garbage collections are necessary.

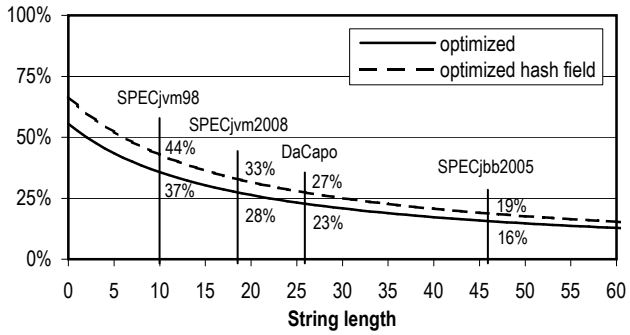

Figure 4: Reduction of memory usage for optimized string objects (higher is better)

- *Faster garbage collection*: An original string is composed of a string object and a character array, both of which must be processed by the garbage collector. The optimized string is a single object and thus reduces the garbage collection time.
- *Better cache behavior*: The two parts of an original string can be spread across the heap, but both parts are always accessed together, which results in a bad cache behavior. The optimized string is one unit that possibly fits into a single cache line.
- *No indirection overhead*: Because of merging the string object with the character array, the characters can be addressed directly and no dereferencing of the character array pointer is necessary.
- *No useless bounds checks*: Original strings perform two bounds checks when they are accessed: an explicit bounds check is implemented in the class `String` because it is a convention to throw a `StringIndexOutOfBoundsException` if it fails. A second implicit bounds check is performed when the character array is accessed. This duplication no longer exists for optimized strings.
- *Faster allocation*: When an original string object and its character array are allocated, both are initialized with default values. Because of a changed allocation of optimized strings, the string characters no longer need to be initialized with default values.
- *No unused characters*: Original string objects may use a far too large character array because of character array sharing. The garbage collector is not aware of this and cannot free any unnecessary memory because it can only determine that the character array is still referenced by a string object. Optimized strings use exactly the minimum necessary amount of memory for storing the characters.

4. Implementation

Our implementation of the optimization uses three new bytecodes for allocating and accessing optimized string objects. These bytecodes are only necessary within the class `String` because the characters of a string are declared as private and cannot be accessed directly from outside.

To introduce the new bytecodes, it is necessary to transform the object code of the class `String`. Although this could be done at run time, it would be complicated. Therefore, we modify the Java compiler (javac) and use it to compile the class `String`. This results in an optimized class, which is created once, and is used

by the VM if the string optimization is enabled. The modified version of javac is not used for compiling any other source code.

Additionally, methods that allocate string objects must be transformed. This must be done at run time because it affects application classes whose source code is not available. The transformation details are explained in Section 4.5.

Introducing new bytecodes for optimized operations inside the VM is a common pattern. These bytecodes use numbers that are unused according to the specification [2]. Because it is only necessary to handle the new bytecode instructions in the interpreter and to support them in the JIT compilers, the impact on the overall VM structure is low.

4.1. Removing the Field `Offset`

To remove the field `offset` of strings, we modify the Java source code of the class `String`. The following three cases must be considered:

- In most cases, it is possible to just remove the accesses to the field `offset`, or to replace them with the constant 0. An example for this case is the method `String.charAt()`.
- Sometimes, the field `offset` is used to implement an optimization such as the sharing of character arrays between string objects. This kind of optimization is no longer possible and it is necessary to create a copy of the characters leading to a certain overhead. An example for this is the method `String.substring()`.
- In rare cases, the string-internal character array is passed to a helper method of another class. The optimized string characters cannot be passed as a character array to a method anymore. It would be necessary to pass the string object itself. To make this possible, the receiving method would have to be overloaded to allow a string object instead of a character array as an argument. We decided to keep the number of changes to a minimum and did not change or add any methods outside of the class `String`. Instead, we copy the string characters to a temporary character array which is then passed as an argument to such methods. This is expensive, but could be easily optimized in the future.

In addition to the Java source code of the class `String`, some parts of the Java HotSpot™ VM must be modified because `String` is a well-known class within the VM. The VM allocates strings for its internal data structures and provides methods to access their content. Also some intrinsic methods use the field `offset` in their handcrafted piece of machine code. The same three cases shown above apply also to the changes inside the VM.

4.2. Character Access

Two new bytecodes are introduced for accessing the characters of an optimized string:

- `scload`: This bytecode loads a string character and is similar to the bytecode `caload` used for loading a character from a character array. `scload` expects two operands on the stack: a reference to the string object and the index of the accessed character.
- `scstore`: This bytecode stores a string character and is similar to the bytecode `castore` used for storing a character in a character array. Compared to the `scload` bytecode, one additional operand is expected on the stack: the character which is to be stored at the specified index of the given string.

Although these bytecodes are similar to the character array access bytecodes, they are still necessary for two reasons:

- As illustrated in Figure 1, the offset of the first character of an optimized string is different from the offset of the first element of a character array. Furthermore, the offset depends on the optimization level: it is 16 if the field `hashcode` is preserved, and 12 if it is removed.

- Each time a character array is indexed, a bounds check is performed. If the index is out of the valid bounds, an `ArrayIndexOutOfBoundsException` is thrown. Optimized string objects also need bounds checks, but within the class `String` it is a convention to throw a `StringIndexOutOfBoundsException` if the check fails.

Introducing the new bytecodes reduces the size of methods in the class `String` because many field loads are no longer necessary. For example, Figure 5 shows the bytecodes of the method `String.charAt()`, whose size is reduced from 33 to 4 bytes. This speeds up the execution and reduces the overall size of the class `String` by approximately 6%.

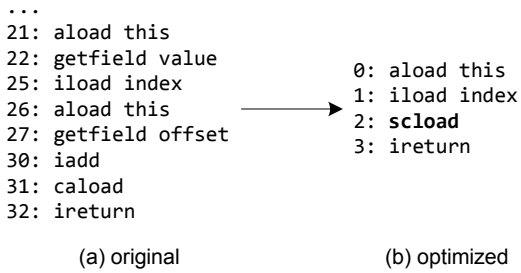

Figure 5: Bytecodes of the method `String.charAt()`

4.3. String Allocation

Allocating optimized string objects is more complicated than accessing their characters. The class `String` currently has 16 constructors. All must be preserved to ensure that no existing program breaks. In Java, the allocation of an object is separated from its initialization, i.e., they are performed by two different bytecodes. For allocating an object, its size must be known. After the allocation, the constructor is invoked to perform the initialization. Arbitrary code can be placed between these two bytecodes, as long as the yet uninitialized object is not accessed. To allocate an optimized string, the number of its characters must be known. However, the number is usually calculated during the execution of the constructor and is therefore not available to the object allocation.

We solve this problem by delaying the string allocation to the point of the initialization. The string constructors are replaced with static factory methods that combine the object allocation and initialization. These factory methods have the same arguments as the constructors. They calculate the length of the resulting string, allocate the optimized string object, and initialize it.

Figure 6 shows the bytecodes available for allocating objects and arrays. For the allocation of an original string object, the bytecode `new` is used. This bytecode can only be used if an object with a statically known size is to be allocated, which applies to all Java objects except arrays. The only operand is an index to a class in the constant pool. When the bytecode is executed, the index is used to fetch a class descriptor that contains the object size. Knowing the size, an object of this class can be allocated.

For allocating arrays, the `newarray` bytecode is used. The array element type is directly encoded in the bytecode and the array length is expected on the operand stack. With the length and the array element type, the total array size is calculated and the allocation is performed.

Optimized string objects have a variable length like arrays, but also fields like objects, so neither of the two previous bytecodes can be used. Therefore, we introduce the bytecode `newstring` that is similar to the bytecode `newarray` and expects the string length on the stack. A bytecode operand, like the element type, is not necessary because the VM knows that a string object only contains characters. Furthermore, the number of fields of a string is fixed and statically known. With this knowledge, the total string size is calculated and the allocation is performed. This bytecode is exclusively used to implement the allocation of optimized string objects within the factory methods.

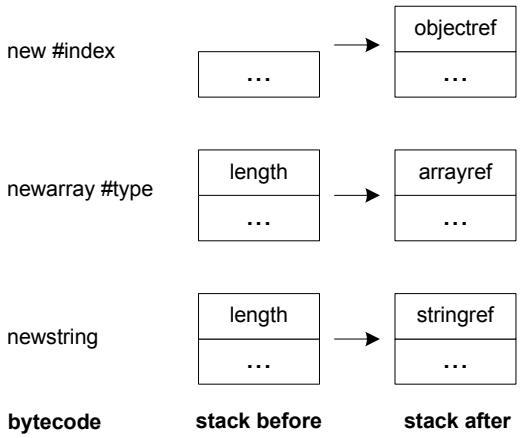

Figure 6: Allocation bytecodes

4.4. Generating the Optimized Class `String`

The Java programming language compiler (javac) is modified to create the class file for the optimized class `String`. The compilation result is packed into a jar file and prepended to the VM's bootstrap classpath to override the default implementation of the class `String`. This modified version of javac is only used to create the optimized string class and is not used for compiling any other Java source code.

Without the modified version of javac, the original class `String` would have to be transformed to the optimized version at run time. This is less flexible and harder to implement. Due to the modified version of javac, it is also possible to provide Java-like source code for the optimized class `String`. This has the advantage that the optimized class can be read and modified more easily. However, the semantics of some statements are adjusted to express the newly introduced bytecodes. The field `value` of the class `String`, which references the character array for original strings, still exists in the Java source code of the optimized class `String`. It represents the string characters that are now embedded in the string object. This field looks and is used like a character array, but actually represents something that cannot be fully expressed in the Java language.

All existing constructors are replaced by the described factory methods. A synthetic constructor is added that has the string length as its only parameter and is used to model the bytecode `newstring` in the Java source code. The modified version of javac uses two additional code generation patterns for generating the optimized class file:

- If the string's field `value` is accessed, the bytecodes `getfield` or `putfield` are omitted. These bytecodes would normally use a string reference that is placed on top of the operand stack. Because they are omitted, the string reference is not consumed and can be used by the character access bytecodes `scload` or `scstore` that are emitted instead of `caload` or `castore`.
- The synthetic constructor, which has the string length as its only parameter, is directly mapped to the bytecode `newstring`. The string length is pushed onto the operand stack like any other constructor parameter. However, the bytecode `newstring` is emitted instead of the constructor invocation.

The result of compiling the method `String.charAt()` with the modified version of javac is shown in Figure 5. The resulting bytecodes for a compilation of a factory method are shown in Figure 7. The synthetic string constructor, used within this method, is compiled to the new string allocation bytecode `newstring` by the modified javac.

4.5. Bytecode Rewriting

The *rewriter* component transforms the original, unoptimized bytecodes to the optimized ones. This is necessary for all methods that allocate string objects. Whenever a class is loaded, the *rewriter* checks if a

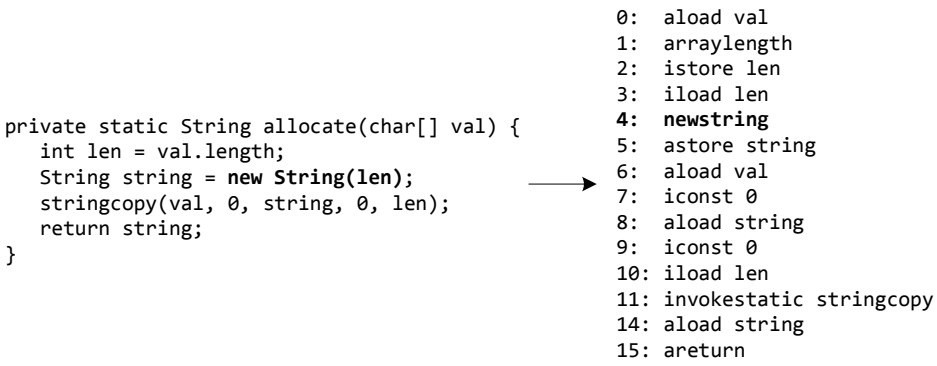

Figure 7: Factory method for string allocation

method of this class allocates string objects. If this is the case, the *rewriter* transforms the bytecodes in three steps, as illustrated in Figure 8:

1. The allocation of the string object, i.e., the bytecode `new`, is removed by replacing it with `nop` bytecodes. The bytecode `new` would push a string reference on the stack, which does not happen anymore because of the removal.
2. The bytecode rewriting process is complicated by stack management instructions like `dup` or `pop` that would use the no longer existing string reference. Each of these management instructions must be modified or removed. Furthermore, some of the subsequent bytecodes might have to be rearranged. Our current implementation is prototypical in that it handles only the most common stack management instructions. Yet, it is complete enough for running nearly all Java programs including the various benchmarks presented in Section 5. Only special bytecode-optimized programs which make use of more sophisticated but rarely used stack management instructions like `dup_x1` might cause problems and are currently not supported, i.e., they are reported as errors.
3. In the last step of the rewriting process, the constructor invocation is replaced with the invocation of the corresponding factory method. Because the factory method has the same arguments as the constructor, no change to the operand stack handling is necessary. This rewriting step can be implemented in two different ways:
   - The constant pool index of the invoked method could be rewritten. This would make it necessary to add the name of the factory method to the constant pool.
   - The method resolution within the VM could be modified. Each time a string constructor must be resolved, the factory method is returned instead. This also works for applications that use the Java Native Interface (JNI) and could not be rewritten otherwise. Therefore, we implemented this approach.

Figure 9 shows a short example for bytecode rewriting. The simple Java method, presented in Figure 9 (a), allocates and returns a string object that is initialized using a character array. The bytecodes of the unoptimized version are shown in Figure 9 (b). The original string object is allocated with the bytecode `new`. In the next step, the string reference is duplicated on the operand stack. This is necessary because the reference is needed both for the invocation of the constructor and the method return. After that, the constructor's parameter is pushed onto the operand stack and the constructor is invoked. It computes the length of the string's character array, allocates it, and initializes it with the characters of the array `ch`.

The *rewriter* component transforms the original bytecodes to the optimized version shown in Figure 9 (c). The transformation happens whenever a method is loaded that allocates string objects. The string allocation and the subsequent reference duplication are replaced with no operation (`nop`) bytecodes. We do not remove these bytecodes completely because this would change the bytecode indices and thus would have side effects on all jumps within the method. If the method is compiled, the `nop` bytecodes are ignored anyway.

```
void rewriteMethod(Method method) {
  foreach(Bytecode code within method) {
    switch(code) {
      case new:
        if(createsString) {
          replaceWithNop();
        }
        break;
      case dup:
        if(isBetweenNewStringAndStringInit) {
          replaceWithNop();
        }
        break;
      case invokespecial:
        if(invokesStringConstructor) {
          replaceWithInvokeFactoryMethod();
        }
        break;
    }
  }
}
```

Figure 8: Bytecode rewriting heuristic

```
public static String createString() {
   char[] ch = ...;
   return new String(ch);
}
```

(a) Java sourcecode

```
...
10: new java.lang.String
13: dup
14: aload ch
15: invokespecial constructor
18: areturn
```

(b) original bytecode

```
...
10: nop
13: nop
14: aload ch
15: invokestatic factory method
18: areturn
```

(c) optimized bytecode

Figure 9: Example for string allocation

4.6. Removing the Field `Hashcode`

When the method `String.hashCode()` is invoked on a string object for the first time, the hash code is computed and cached in the field `hashcode` to avoid multiple computations. The hash code is computed from the string characters, i.e., two string objects with the same contents have the same hash code. The field `hashcode` requires four bytes per string object, so its removal is beneficial. However, recomputing the hash code each time it is accessed would slow down many applications.

We use the part of the mark word in the object header where normally the identity hash code is stored to cache the string hash code. As it is not guaranteed that the identity hash code is unique for every object, it should not have any side effects if the identity hash code of strings is equal to the string hash code. As long as `String.hashCode()` is executed by the interpreter, the hash code is not cached and is calculated upon each method invocation. This is necessary because the Java object header cannot be accessed via a bytecode. When the method `String.hashCode()` is passed to the JIT compiler, it is not compiled but an intrinsic method is used that calculates the hash code once and caches its value in the object header. The actual intrinsic method is written in assembler and performs the steps shown in Figure 10. If the object is unlocked, its mark word is accessed to extract the possibly cached hash code. If the object is either locked or its hash code was not cached yet, we compute the hash code. This computed hash code is then cached in the mark word if the object is unlocked. The hash code is not cached if the object is locked because the mark word points to a locking data structure when this code path is taken. This is a rare case because strings are normally not used for synchronization.

As mentioned before, the hash code is truncated to 25 bits on 32-bit architectures when it is stored in the mark word. Although this should not have a significant negative effect, the hash code algorithm is specified in the documentation of the class `String`. Therefore, this optimization might violate the specification on 32-bit architectures. Because of this, the next section evaluates our optimization with and without the elimination of the field `hashcode`. On 64-bit architectures, this optimization complies with the specification because the object header is large enough to hold the hash code without truncation.

4.7. Further Adjustments

Some existing optimizations are voided by the new optimized class `String`. The string characters can no longer be copied using the method `System.arraycopy()` because they are not stored as a real array

```
int hashCode() {
  int hashcode = 0;

  if(this.isUnlocked()) {
    hashcode = extractHashCode(this.markWord);
  }

  if(hashcode == 0) {
    hashcode = computeHashCode(this);

    if(this.Unlocked()) {
      cacheHashCode(this, hashcode);
    }
  }

  return hashcode;
}
```

Figure 10: `String.hashCode()` intrinsic method

anymore. `System.arraycopy()` is faster than a loop in Java code because some bounds and type checks can be omitted. Therefore, we use specialized versions of `System.arraycopy()` for optimized string objects. These methods copy a range of characters between two strings or between a string and a character array. In the JIT compiler, these new methods are handled as intrinsic methods and share nearly the whole code with the method `System.arraycopy()`.

4.8. Implementation Details

For our optimization we needed to modify the size computation of objects. During a garbage collection, the garbage collector iterates over all objects on the heap. For this iteration, the size of each object on the heap must be computed. Because a large heap offers enough space for millions of objects, the necessary time for computing an object's size must be as low as possible. The simplest implementation of the object size computation would call a virtual method of the class descriptor object. However, this is too slow for a large number of invocations because calling a virtual method adds an additional indirection step.

The Java HotSpot^{TM} VM implements a fast variant for the size computation of objects, which uses the so-called *layout helper* field of class descriptor objects. This field only stores a useful value for objects that are directly allocated from Java source code, i.e., Java class instances and arrays, as those are the majority on the heap. For all other VM internal objects, a virtual method of the class descriptor object is called. The layout helper of Java class instances contains the statically known object size. For arrays, the size is computed using the element type and the number of elements.

The string optimization causes some problems with this existing infrastructure because optimized strings are Java class instances that do not have a statically known size. Therefore, distinguishing between Java class instances, arrays and other objects is no longer sufficient. To support a fast size computation for optimized string objects, we modify the layout helper to distinguish between objects with a fixed size, objects with a computed size, and other objects. We also provide the possibility to use the fast size computation for VM internal objects. The changes of the layout helper require several modifications of other parts of the Java HotSpot^{TM} VM because the layout helper is also used in platform-dependent code for the allocation of Java class instances and arrays.

The implementation of our optimization within the interpreter and the JIT compilers mainly involves the following changes:

- *Changes to the VM infrastructure*: Elimination of the string field `offset` and the changes of the layout helper require modifications of the Java HotSpot^{TM} VM infrastructure including platform-dependent code. Several parts of these changes affect code that is shared between the interpreter and the JIT compilers.

- *Implementing the new bytecodes*: The newly introduced bytecodes are similar to existing bytecodes. So, we reused as much from the existing infrastructure as possible. However, the interpreter and the two JIT compilers all had to be changed separately as they use different approaches and different code bases. The following list exemplarily summarizes the necessary implementation steps for the client compiler:
  - The allocation of optimized string objects requires one new HIR instruction. For accessing the characters within an optimized string, it is sufficient to extend the HIR instruction used for accessing a character array. This has the advantage that existing optimizations, such as bounds-check elimination, are automatically applied to optimized strings.
  - For removing the field `hashcode`, one LIR instruction was introduced that represents the intrinsic method `String.hashCode()`. All other required LIR functionality is mapped to sequences of already existing instructions.

4.9. Compatibility Issues

Although the implementation of the string optimization is complete enough for running nearly all Java programs, it has some remaining compatibility issues with the Java specification. If this optimization is to be integrated in a product version of the Java HotSpot™ VM, the following issues must be addressed:

- *Reflection*: The optimization replaces the `String` constructors with factory methods. Therefore, no constructors can be accessed via reflection. This could be solved by returning the appropriate factory method whenever a constructor is accessed via reflection. Some additional changes in the Java HotSpot™ VM would be required for that. Existing Java code may also access the fields `offset`, `count`, `hashcode`, and `value` of the class `String` via reflection. These fields should not cause severe problems as they are all declared as private. Code that accesses private fields via reflection should be aware of possible implementation changes.

- *Exception handling*: Exceptions that occur during the allocation or initialization of optimized strings are thrown in the context of the factory methods. However, the programmer expects the exceptions to be thrown in the constructor as the factory methods are only used internally and are not visible to the programmer.

- *JNI*: Nearly all compatibility issues with the JNI were solved by modifying the method resolution. However, the JNI provides the method `AllocObject` that allocates an object without invoking its constructor. For optimized string objects, this is no longer possible because the factory methods combine the object allocation and initialization. Therefore, this method would have to execute a code similar to the bytecode `newstring`.

- *Bytecode rewriting*: The rewriting heuristic implemented for the prototype must be extended to fully comply with the Java specification. It is necessary to parse the bytecodes and to build a representation where stack management instructions such as `dup_x1` can be safely reorganized or deleted.

5. Evaluation

Our string optimization is integrated into Sun Microsystems' Java HotSpot™ VM, using the early access version b24 of the upcoming JDK 7. The benchmarking system has the following configuration: an Intel Core2 Quad processor with 4 cores running at 2.4 GHz, 2 * 2 MB L2 cache, 2 GB main memory, and with Windows XP Professional as the operating system. For measuring the performance and the number of allocated bytes, we use the benchmarks SPECjvm98 [4], SPECjvm2008 [6], SPECjbb2005 [3], and DaCapo [5]. We present the results of four different configurations:

- Our baseline configuration is the unmodified Java 7 build b24.

- In the "optimized" configuration, all optimizations described in this paper except the removal of the field `hashcode` are performed.
- Our optimization has two benefits: a reduced memory consumption because of the smaller string objects, and a reduced number of memory accesses because of the eliminated fields. The "overhead" configuration increases the size of the optimized string objects artificially to the size of the original strings by adding a padding of 20 bytes per string. If a benchmark does not invoke any methods that use character array sharing for the original strings, this configuration allocates exactly the same number of bytes as the baseline. Any additional memory overhead, e.g., in Figure 16, is a result of the missing character array sharing. Therefore, this configuration indicates how frequently character array sharing is used in a benchmark. The performance loss in comparision with the "optimized" configuration shows the impact of the reduced memory usage on the performance.
- The "optimized hash field" configuration uses all optimizations described in this paper including the optimization of the field `hashcode`. We use a 32-bit architecture for benchmarking and therefore the hash code is truncated to 25 bits. Because the SPECjbb2005 benchmark compares the hash code of a string object to a hardcoded value during the startup, the benchmark needed to be modified slightly.

5.1. SPECjbb2005

The SPECjbb2005 benchmark represents a client/server business application. All operations are performed on a database that is held in the physical memory. With an increasing number of warehouses, the size of the database increases and less memory is available for executing transactions on the warehouses. Therefore, the number of garbage collections increases, which has a negative impact on the performance.

The benchmark result is the total throughput in so called SPECjbb2005 business operations per second (bops). This metric is calculated from the total number of executed transactions on the database. In this benchmark, a high number of string operations is performed. Unless stated otherwise, a heap size of 1200 MB is used for all measurements.

Figure 11 illustrates the SPECjbb2005 performance and the average amount of used memory after a full garbage collection. The used memory after a full garbage collection serves as an approximation of the application's minimum heap size. Both numbers are significantly improved by our string optimization.

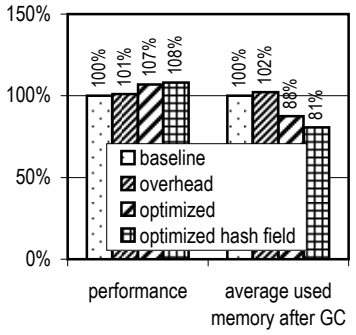

Figure 11: SPECjbb2005: performance and memory usage

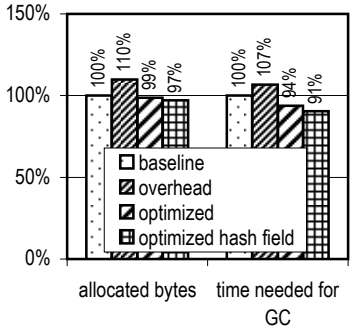

Figure 12: SPECjbb2005: memory usage for a fixed number of transactions (lower is better)

The SPECjbb2005 benchmark always runs 240 seconds for a specific number of warehouses. Therefore, the total number of allocated bytes depends on the performance, i.e., on how many transactions can be run in this time frame. To measure the number of allocated bytes independently of the performance, we used a slightly modified version of the SPECjbb2005 benchmark that executes a fixed number of transactions on four warehouses. For the optimized configurations, the number of allocated bytes is reduced as shown in Figure 12. Furthermore, the optimizations also reduce the time necessary for garbage collection. The configuration "overhead" allocates more bytes than the baseline because original strings use character array sharing for explicit string copying.

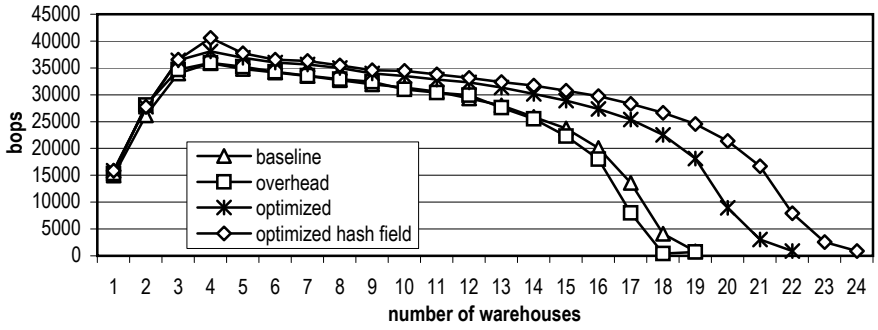

Figure 13: SPECjbb2005: performance for various numbers of warehouses (higher is better)

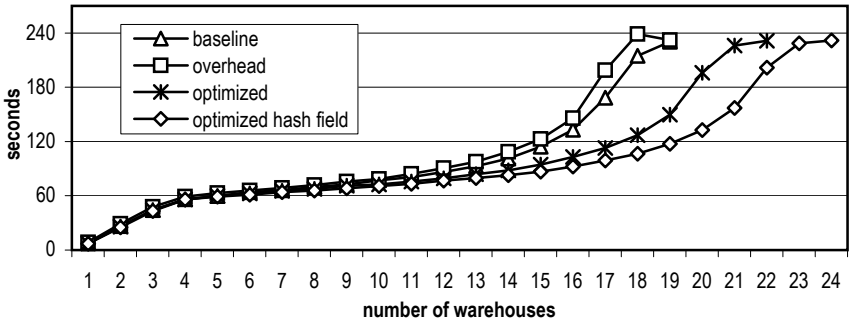

Figure 14: SPECjbb2005: garbage collection time for various numbers of warehouses (lower is better)

To further evaluate the impact of the reduced memory consumption, we executed each configuration with a heap size of 512 MB, up to the number of warehouses where an `OutOfMemoryException` is thrown. Figure 13 shows the performance for various numbers of warehouses. The performance increases up to 4 warehouses because each warehouse uses its own thread and the benchmarking system has 4 cores. With a higher number of warehouses, the thread overhead and the memory usage increases, so that the performance decreases. Because of the reduction of memory usage, it is possible to execute the SPECjbb2005 benchmark with 24 instead of 19 warehouses.

Figure 14 shows the overall time spent in the garbage collector for runs with various numbers of warehouses. The configurations that reduce the memory usage also spend less time in the garbage collector. The memory-resident database uses a smaller part of the heap and therefore more memory is available for the actual execution, which then needs fewer garbage collections. There is a clear correlation between Figure 13 and Figure 14: the performance decreases as the time for garbage collection increases.

5.2. DaCapo

The DaCapo benchmark suite consists of eleven object-oriented applications. We used the release version 2006-10-MR2 and executed each benchmark five times, so that the execution time converges because all relevant methods have been compiled by then. We present the slowest and the fastest run for each benchmark. The slowest run, which is always the first one in our case, shows the startup performance of the JVM, while the fastest run shows the achievable peak performance (all relevant methods compiled). Furthermore, the geometric mean of all results is presented. A heap size of 256 MB is used for all benchmarks.

The performance for the benchmarks in the DaCapo suite is presented in Figure 15. In this diagram, the slowest and the fastest runs for each benchmark are shown on top of each other. Both runs are shown relative to the fastest run of the baseline. The light bars refer to the slowest runs, the dark bars to the fastest runs.

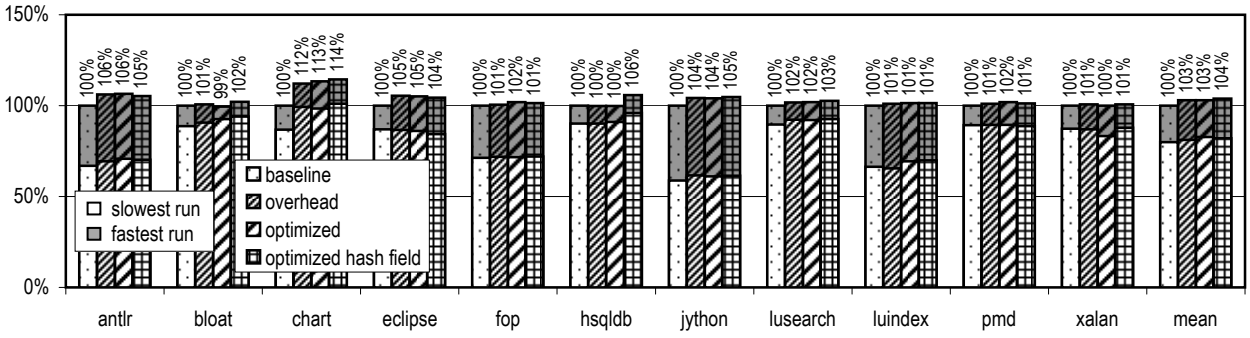

Figure 15: DaCapo: performance results (higher is better)

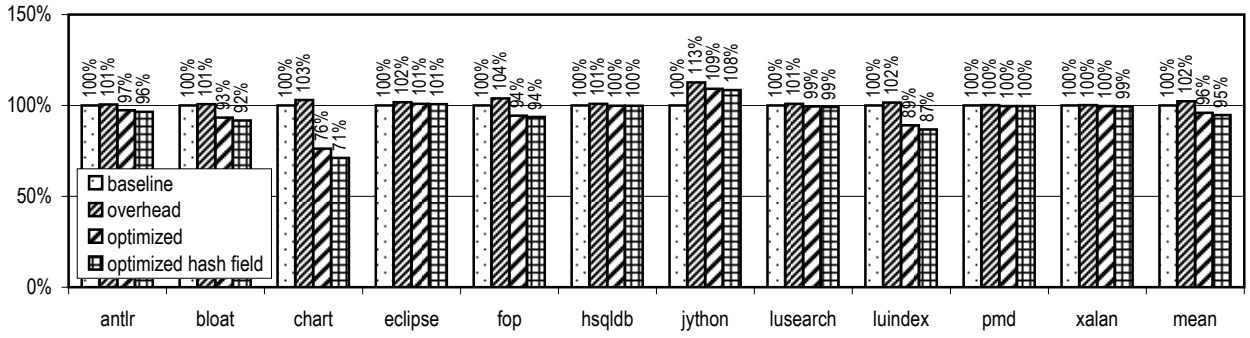

Figure 16: DaCapo: number of allocated bytes (lower is better)

Both the fastest and the slowest runs are improved for nearly all benchmarks. Especially the chart benchmark, which is string-intensive, profits greatly from the string optimization. Other benchmarks with a considerable speedup are antlr, hsqldb, and jython. Benchmarks that use only few strings show neither a speedup nor a slowdown. For most benchmarks the configuration "overhead" shows a similar performance as the configuration "optimized". Therefore, the reduction of the memory usage has little positive effect on the performance of these benchmarks.

The number of allocated bytes for each benchmark is presented in Figure 16. Again, the chart benchmark profits greatly from the optimization and allocates less memory. This benchmark also shows the impact of the elimination of the field `hashcode`. While the number of allocated bytes is reduced for most string-intensive benchmarks, the jython benchmark shows a contrary result. More memory must be allocated for this benchmark because jython uses character array sharing, which is no longer possible with optimized string objects. Therefore, new string objects are allocated and the characters must be copied. Nevertheless, the performance still shows a speedup. The configuration "overhead" allocates more memory than the baseline for most benchmarks. This indicates that a significant amount of character array sharing is used.

5.3. SPECjvm98

The SPECjvm98 benchmark suite contains seven benchmarks derived from typical client applications. Similar to the DaCapo benchmark suite, we executed each benchmark until the execution time converged. We report the slowest and the fastest run for the string-intensive benchmarks db, jack, and javac (no significant difference to the baseline is measured for the other four benchmarks), as well as the geometric mean of all seven benchmarks. A heap size of 64 MB is used for all benchmarks.

Figure 17 illustrates the results of the SPECjvm98 benchmark suite. In this diagram, the slowest and the fastest run for each benchmark are shown on top of each other. Both runs are shown relative to the fastest run of the baseline and the light bars refer to the slowest runs, the dark bars to the fastest runs.

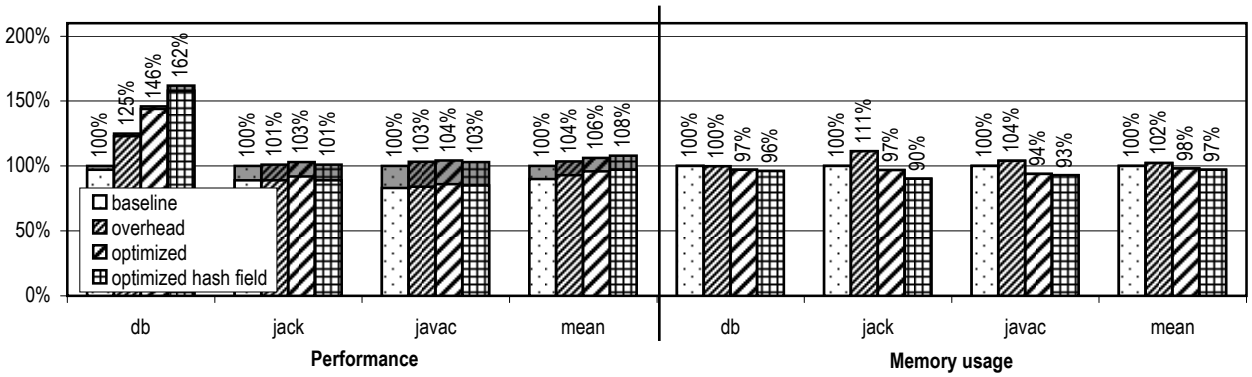

Figure 17: SPECjvm98: performance and allocated bytes for string-intensive benchmarks

The db benchmark shows an exceptionally high speedup. Other than the name suggests, this benchmark spends most time sorting a list of strings and is therefore the ideal target for our optimization. The number of allocated bytes decreases only slightly, so the high speedup results from the removal of the field accesses and the better cache behavior. For all benchmarks the performance of the slowest run as well as of the fastest run is greater or equal to the baseline. Furthermore, the number of allocated bytes is smaller or equal to the baseline for all benchmarks. This means that the performance and the number of allocated bytes are optimized without any negative effect on any of the benchmarks. The largest reduction of the number of allocated bytes is measured for the jack benchmark.

5.4. SPECjvm2008

The SPECjvm2008 benchmark suite contains ten benchmarks, some of them consisting of several sub-benchmarks. It is designed to replace the SPECjvm98 benchmark suite. The sub-benchmarks compiler.compiler and compiler.sunflow compile Java source code to bytecode. Compiling against the optimized class `String` would cause compilation errors whenever a string constructor is invoked explicitly. This is a limitation of our current implementation (see Section 4.9). To avoid the compilation errors, we added the original source code of the class `String` to the compilation set of both sub-benchmarks. This modified compilation set is used for all benchmark runs.

Many of the benchmarks perform only numerical computations without using strings. For this reason, we omitted two larger groups of sub-benchmarks: the three crypto benchmarks, and the nine scimark benchmarks. All of these show the same performance in all configurations. A heap size of 512 MB is used for all benchmarks. Figure 18 shows the results of the SPECjvm2008 benchmark suite. Mainly the two xml sub-benchmarks, which are string-intensive, profit from the optimization.

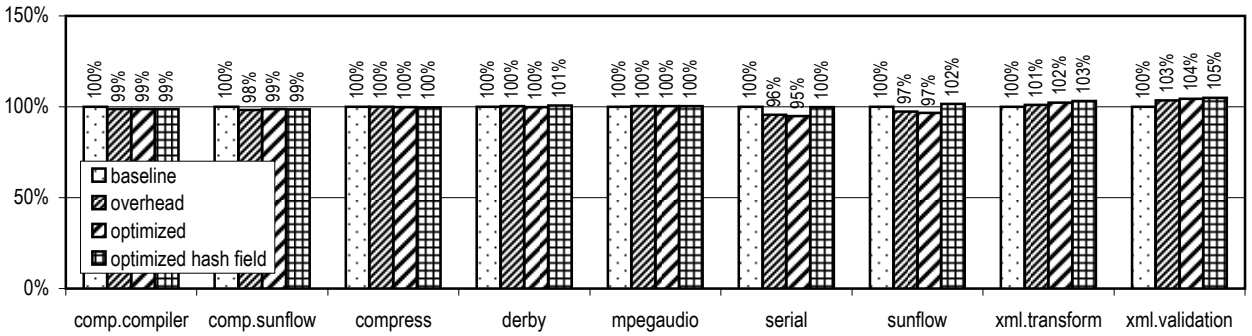

Figure 18: SPECjvm2008: performance results (higher is better)

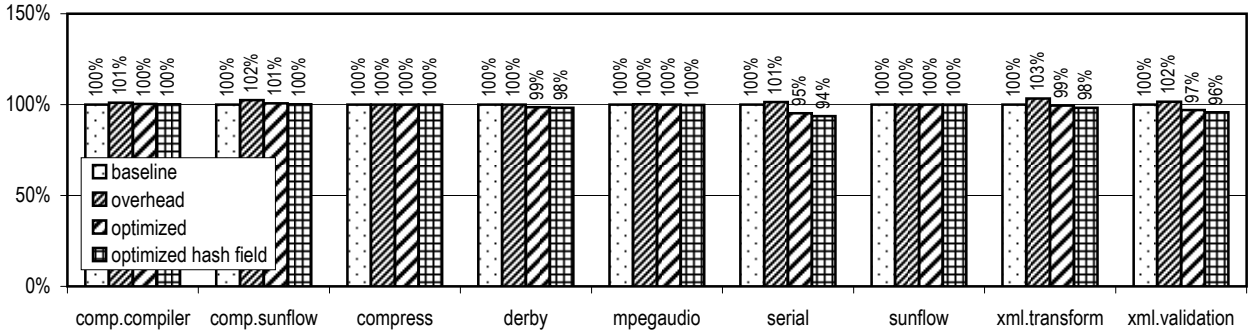

Figure 19: SPECjvm2008: number of allocated bytes (lower is better)

Similar to the SPECjbb2005 benchmark, the SPECjvm2008 benchmark suite runs every sub-benchmark for 240 seconds. We use the SPECjvm2008 fixed size workload Lagom to ensure that the number of allocated bytes does not depend on the performance. As shown in Figure 19, the number of allocated bytes is reduced for the string-intensive benchmarks. The configuration "overhead" allocates more bytes than the baseline, which indicates that original strings profit from character array sharing. Mainly the benchmarks serial, xml.transform and xml.validation profit from the optimization and allocate less memory.

5.5. Server Compiler

To evaluate our optimization with the server compiler, we use the same four benchmark suites. As in the previous sections, we omit some benchmarks from SPECjvm98 and SPECjvm2008 that do not use strings. Our prototype implementation for the server compiler is only optimized for the case where a string object has the same memory layout as a character array, i.e., the configuration *optimized hash field*. This allowed us to re-use several optimized copying and access methods. Figure 20 shows the performance results when our optimization is applied to the server compiler.

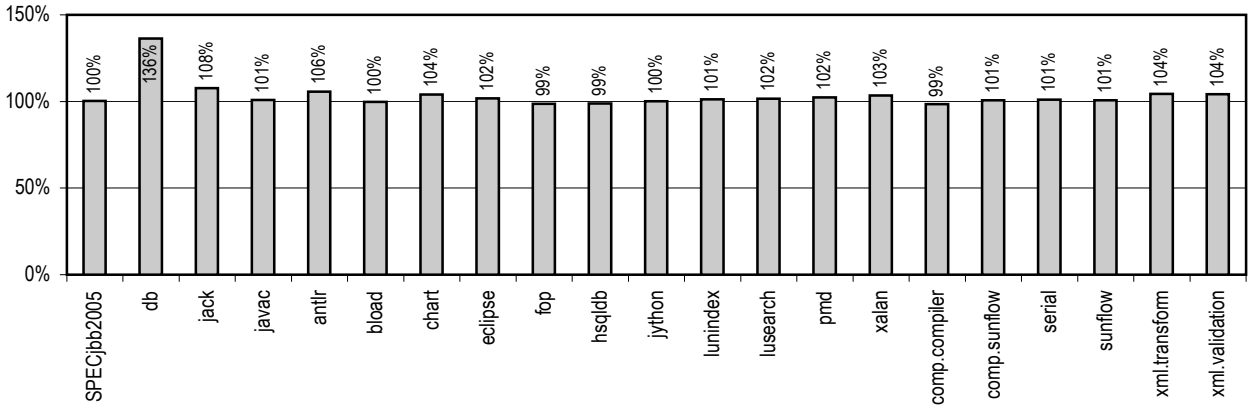

Figure 20: Server compiler: performance of SPECjbb2005, SPECjvm98, DaCapo, and SPECjvm2008 (higher is better)

Many benchmarks profit significantly from our optimization, especially the SPECjvm98 benchmarks db and jack, the DaCapo benchmarks antlr, chart, and xalan, as well as the SPECjvm2008 xml benchmarks. In general, the speedup for the server compiler is smaller than the speedup for the client compiler. There are several reasons for this. First, the server compiler optimizes field accesses more aggressively and can therefore eliminate some disadvantages of unoptimized strings. Secondly, the server VM uses a parallel garbage collector with a copying order where unoptimized string objects and their character arrays tend to be placed consecutively in memory. So, optimized strings have only a slightly better cache behavior.

Additionally, some compiler optimizations are not yet supported for our modified string objects, for example escape analysis.

With a large heap size, the SPECjbb2005 benchmark results for the configurations baseline and "optimized hash field" are equal. Figure 21 shows the performance for these two configurations with a heap size of 512 MB and an increasing number of warehouses. Due to the smaller heap size and the larger number of warehouses, the figure clearly shows a speedup for our optimization. This is the advantage of the reduced memory usage.

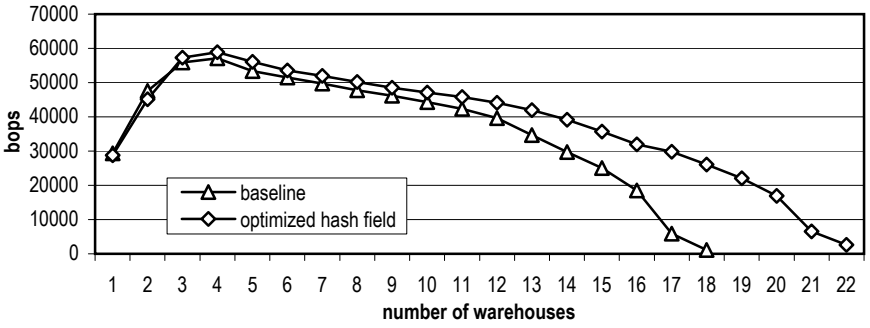

Figure 21: SPECjbb2005: performance for various numbers of warehouses with the server compiler (higher is better)

5.6. Further Evaluations

The string optimization has a negative impact on some methods such as `String.substring()`. Therefore, we executed our own micro benchmark to determine the negative impact on such methods. This micro benchmark is a worst-case scenario for our string optimization and invokes the method `String.substring()` on random strings with a length of 0 to 100 characters. For this micro benchmark, the optimized configurations are about 25% slower than the baseline.

About 19% of all string allocations in the SPECjbb2005 benchmark are explicit string copies allocated with the constructor `String(String)`. For original strings that do not use their full character array, this constructor allocates a trimmed copy of the character array. Because optimized strings always use all their characters, explicit copies are useless. However, they cannot be removed easily for two reasons:

- *Object equality*: Without an explicit string copy, the semantics of object equality checks can change.
- *Synchronization*: When a string object is used as a monitor, the program behavior might change if no explicit string copy is allocated.

These cases would have to be detected to safely eliminate explicit string copying. Both cases do not apply to any of the explicit string copies in the SPECjbb2005 benchmark. To measure which performance could be expected from the string optimization if the programmer knows that the allocation of explicit string copies is unnecessary, all explicit string copies were removed for the SPECjbb2005 benchmark. Due to the factory methods, this can be achieved easily by modifying the factory method `String allocate(String original)` to be an identity function, i.e., to return `original` instead of a newly allocated copy. In comparison to the baseline, the "optimized hash field" configuration shows a 18% higher performance and a reduction of the average used memory after a full garbage collection by 20%. Furthermore, the number of allocated bytes is reduced by 11%, and the time necessary for garbage collection is reduced by 30%.

6. Related Work

Boldi et al. implemented a class `MutableString` that combines the advantages of the classes `String` and `StringBuffer` [15]. A `MutableString` can be in the state "compact" or "loose". Depending on this

state, the `MutableString` has either the advantages of the class `String` or `StringBuffer`. If the capacity of the string is no longer sufficient because of string concatenation, and the string's state is "compact", the `MutableString` is resized to exactly fit the content. If it is in the state "loose", the size of the character array is doubled. To allow subclasses, the class is not final and it allows the direct access to the character array without any restrictions. This direct access is potentially unsafe and must be used carefully by the programmer. In contrast to our optimization, existing programs must be changed and recompiled to use the advantages of this new class. Furthermore, their optimization does not reduce the memory usage.

Tian addressed the performance problem of string concatenations [16]. If two string objects are concatenated, a new temporary `StringBuilder` or `StringBuffer` object is allocated, to which the characters of both strings are copied. On this temporary object, the method `toString()` is invoked to allocate the resulting string, which again copies all characters. Using the Java bytecode optimization framework Soot [17], a bytecode transformation was implemented that removes redundant buffer allocations and reuses existing buffers for the concatenation. With this transformation, the performance of string concatenation is improved nearly up to the performance of the class `StringBuilder`. This optimization must be applied directly to the class file, while ours is performed automatically behind the scenes by the VM and covers more than string concatenations.

Ananian et al. implemented several techniques to reduce the memory usage of object-oriented programs [18]. These techniques include field reduction and the elimination of unread or constant fields. The value range for each field is analyzed to replace the data type with a less space consuming one. Furthermore, static specialization is used to create two different string classes: one string class without the field `offset` (`SmallString`) and one with the field `offset` (`BigString`). If the `offset` is zero, a `SmallString` is allocated, otherwise a `BigString`. An evaluation with the SPECjvm98 benchmark suite showed that the maximum live heap size is reduced by up to 40%. Some of the optimizations have a negative impact on the performance, which manifests in a performance change from -60% to +10% for the SPECjvm98 benchmark suite. We optimize only string objects, but always remove the field `offset` and merge the string's character array with the string object to save further memory.

Chen et al. implemented two different approaches to reduce the memory consumption. A heap compression algorithm was implemented that targets memory-constrained environments such as mobile devices. It reduces the minimum heap size that is necessary for the execution of an application [19]. A Mark-Compact-Compress-Lazy Allocate garbage collector is its basic component. To reduce the compression and decompression overhead, large instances and arrays are split into multiple objects that can be compressed and decompressed independently. Lazy allocation delays the allocation of parts of large arrays that are not used immediately. For the heap compression, an arbitrary compression algorithm can be used. The evaluation was done with a "zero removal" compression algorithm, and showed that 35% of memory can be saved on average. In contrast to our optimization, the compression can be applied to all kinds of Java objects but has a negative impact on the performance.

The second approach exploits frequent field values to reduce the memory usage [20]. It uses the fact that a small number of distinct values appear in lots of fields. Based on this, two object compressions are proposed to reduce the memory usage. The first one is specialized on fields that are zero or null. The other one is used for fields with other frequent values. To determine fields that can be optimized, the application is executed with various inputs. This information is written in a separate description file that the JVM uses to perform the optimization. Depending on the program inputs, the description file varies, which affects the performance and the memory usage differently. The evaluation with the SPECjvm98 benchmark suite shows that the minimum heap size is reduced by up to 24% (14% on average). The loss of performance is below 2% for most cases. Our optimization focuses on strings but shows a reduction of memory usage and a speedup for string-intensive benchmarks.

Dolby et al. implemented object inlining in a static compiler for a dialect of C++ [21]. This optimization can merge some referenced objects with the referencing one, and is not limited to string objects. This improves the cache behavior and reduces the indirection overhead. A field is a candidate for inlining when all its uses can be located and when the relationship between the parent and the child object is unambiguous. For this, a global data flow analysis is necessary that takes up to half of the compilation time. The average speedup for the C++ benchmarks is 10% with a maximum of 50%. In contrast to our optimization, object

inlining is not limited to string objects, but our optimization is performed at run time in a virtual machine and has only a negligible analysis overhead.

Wimmer et al. implemented object inlining for the Java HotSpot™ VM [22, 23]. To determine fields worth inlining, read barriers are used. The JIT compiler ensures that the candidate field is not overwritten and that the objects are allocated together. This is necessary because the referencing and the referenced objects must be located next to each other in the heap. Therefore, the garbage collector was also modified to ensure that inlined objects are not separated. The optimization is performed automatically at run time, but cannot optimize strings because a character array can be referenced by multiple string objects. The mean peak performance of the SPECjvm98 benchmark suite is improved by up to 51% (9% on average).

Oracle (formerly BEA Systems, Inc.) has a patent pending to address the inefficiency of `StringBuilder` and `StringBuffer` operations [24]. When a string is appended to a `StringBuilder` or `StringBuffer`, all characters are copied to the buffer's character array. If the size of this character array is no longer sufficient, a larger character array is allocated, to which all characters must be copied. When the method `toString()` is invoked, the characters are copied another time to the resulting string object. Therefore, it is more efficient to store the references to the appended string objects instead of copying the characters to a buffer. Additionally, a larger variety of `append()` methods reduces the number of method invocations. When the `toString()` method is invoked, an appropriate character array with the size of the summed up length of all string parts can be allocated. All characters are copied to this character array, which is then referenced by the resulting string object. This optimization is beneficial for string concatenation, while our optimization is advantageous to the usage of string objects in general.

Zilles implemented accordion arrays for Java to reduce the memory usage of character arrays [25]. Java characters arrays are always Unicode-based, even if the top bytes of the characters are zero. Storing these characters as bytes instead of Unicode characters saves 50% of the memory. However, code that accesses character arrays must determine dynamically if the array uses one or two bytes for storing each character. If a Unicode character is to be stored in an array that uses only one byte for storing each character, the array must be inflated. The performance improves by 8% for the SPECjbb2005 benchmark and by 2% for the DaCapo benchmark. The size of the live objects is reduced by up to 40%. By reducing the memory usage of character arrays, memory is also saved for string objects. We do not compress any characters but merge the string object and the character array to improve the performance and to reduce the memory usage.

Shuf et al. distinguished between frequently allocated (prolific) and rarely allocated (non-prolific) types. Based on this, several optimizations were implemented for the Jalapeño VM. A type-based garbage collector that distinguishes between prolific and non-prolific types increases the data locality and reduces the garbage collection time by up to 15% [26]. Additionally, a short type pointer technique eliminates the one machine word type pointer for prolific types by adding some small type information to the mark word. This reduces the memory requirements by up to 16%. Furthermore, two approaches to improve the locality of Java applications were implemented [27]. The first one allocates prolific objects that are connected by references next to each other in the memory. The second approach uses locality based graph traversal to reduce the garbage collection time and to increase the locality. Benchmark results for SPECjvm98, SPECjbb2000, and jOlden show that a combination of both approaches improve the performance by up to 22% (10% on average), if a non-copying mark-and-sweep garbage collector is used. Our optimization improves the locality for string objects by merging the string object with its character array. This also reduces the garbage collection time and the memory usage.

Casey et al. introduced new bytecodes to increase the performance of a Java interpreter [28]. Because some operands are frequently used for specific bytecodes, specialized bytecodes with hardwired operands reduce the required operand fetching. Furthermore, additional bytecodes are introduced that combine common bytecode sequences. Instruction replication is another technique that uses multiple implementations for one bytecode to reduce indirect branch mispredictions. Which implementation of the bytecode is executed depends on the subsequent bytecodes. Results for the SPECjvm98 benchmark show an average speedup of 30% to 35%. We also use introduce new bytecodes for our optimization but those are required for the implementation and are not mainly used to improve the performance.

7. Conclusions

We presented a string optimization that is performed automatically at run time by the Java HotSpot™ VM. The string object and the character array of an original string are merged into a single object. For this merging, new bytecodes are introduced that are only used within the class `String`. A modified version of javac generates the optimized class `String` that uses the newly introduced bytecodes. All methods that allocate string objects are rewritten once at run time during class loading. The merging removes additional field accesses, reduces the memory usage, speeds up garbage collection, and leads to a better cache behavior. The evaluation with several benchmark suites shows that these advantages result in a significantly higher overall performance and a lower memory usage.

Acknowledgements

We would like to thank current and former members of the Java HotSpot™ compiler team at Sun Microsystems, especially Thomas Rodriguez, David Cox, and Kenneth Russell, for their persistent support, for contributing many ideas and for helpful comments on all parts of the Java HotSpot™ VM. We are especially grateful that Thomas Rodriguez ported our implementation for the client compiler to the server compiler.

References

[1] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java™ Language Specification, 3rd Edition, Addison-Wesley, 2005.

[2] T. Lindholm, F. Yellin, The Java™ Virtual Machine Specification, 2nd Edition, Addison-Wesley, 1999.

[3] Standard Performance Evaluation Corporation, The SPECjbb2005 Benchmark, http://www.spec.org/jbb2005/ (2005).

[4] Standard Performance Evaluation Corporation, The SPECjvm98 Benchmarks, http://www.spec.org/jvm98/ (1998).

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 2006, pp. 169–190. doi:10.1145/1167473.1167488.

[6] Standard Performance Evaluation Corporation, The SPECjvm2008 Benchmarks, http://www.spec.org/jvm2008/ (2008).

[7] C. Häubl, C. Wimmer, H. Mössenböck, Optimized strings for the Java HotSpot™ virtual machine, in: Proceedings of the International Symposium on Principles and Practice of Programming in Java, ACM Press, 2008, pp. 105–114. doi:10.1145/1411732.1411747.

[8] Sun Microsystems, Inc., Java Platform, Standard Edition 7 Source Snapshot Releases, http://download.java.net/jdk7/ (2007).

[9] R. Griesemer, S. Mitrovic, A compiler for the Java HotSpot™ virtual machine, in: L. Böszörményi, J. Gutknecht, G. Pomberger (Eds.), The School of Niklaus Wirth: The Art of Simplicity, dpunkt.verlag, 2000, pp. 133–152.

[10] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, D. Cox, Design of the Java HotSpot™ client compiler for Java 6, ACM Transactions on Architecture and Code Optimization 5 (1) (2008) 7. doi:10.1145/1369396.1370017.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (4) (1991) 451–490. doi:10.1145/115372.115320.

[12] C. Wimmer, H. Mössenböck, Optimized interval splitting in a linear scan register allocator, in: Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments, ACM Press, 2005, pp. 132–141. doi:10.1145/1064979.1064998.

[13] M. Paleczny, C. Vick, C. Click, The Java HotSpot™ server compiler, in: Proceedings of the Java Virtual Machine Research and Technology Symposium, USENIX, 2001, pp. 1–12.

[14] K. Russell, D. Detlefs, Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2006, pp. 263–272. doi:10.1145/1167515.1167496.

[15] P. Boldi, S. Vigna, Mutable strings in Java: Design, implementation and lightweight text-search algorithms, Science of Computer Programming 54 (1) (2005) 3–23. doi:10.1016/j.scico.2004.05.003.

[16] Y. H. Tian, String concatenation optimization on Java bytecode, in: Proceedings of the Conference on Programming Languages and Compilers, CSREA Press, 2006, pp. 945–951.

[17] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, Soot – A Java optimization framework, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, 1999, pp. 125–135.

[18] C. S. Ananian, M. Rinard, Data size optimizations for Java programs, in: Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems, ACM Press, 2003, pp. 59–68. doi:10.1145/780732.780741.
[19] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, M. Wolczko, Heap compression for memory-constrained Java environments, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 2003, pp. 282–301. doi:10.1145/949305.949330.
[20] G. Chen, M. Kandemir, M. J. Irwin, Exploiting frequent field values in Java objects for reducing heap memory requirements, in: Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments, ACM Press, 2005, pp. 68–78. doi:10.1145/1064979.1064990.
[21] J. Dolby, A. Chien, An automatic object inlining optimization and its evaluation, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 2000, pp. 345–357. doi:10.1145/349299.349344.
[22] C. Wimmer, H. Mössenböck, Automatic feedback-directed object inlining in the Java HotSpot™ virtual machine, in: Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments, ACM Press, 2007, pp. 12–21. doi:10.1145/1254810.1254813.
[23] C. Wimmer, H. Mössenböck, Automatic array inlining in Java virtual machines, in: Proceedings of the International Symposium on Code Generation and Optimization, ACM Press, 2008, pp. 14–23. doi:10.1145/1356058.1356061.
[24] S. Larsen, What's Hot in BEA JRockit, BEA Systems, Inc., http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2171.pdf (2007).
[25] C. Zilles, Accordion arrays, in: Proceedings of the International Symposium on Memory Management, ACM Press, 2007, pp. 55–66. doi:10.1145/1296907.1296916.
[26] Y. Shuf, M. Gupta, R. Bordawekar, J. P. Singh, Exploiting prolific types for memory management and optimizations, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 2002, pp. 295–306. doi:10.1145/503272.503300.
[27] Y. Shuf, M. Gupta, H. Franke, A. Appel, J. P. Singh, Creating and preserving locality of Java applications at allocation and garbage collection times, in: Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, 2002, pp. 13–25. doi:10.1145/582419.582422.
[28] K. Casey, M. A. Ertl, D. Gregg, Optimizations for a Java Interpreter Using Instruction Set Enhancement, Tech. rep., Department of Computer Science, University of Dublin, Trinity College (2005).