JOHANNES KEPLER
UNIVERSITY LINZ | JKU

Christian Häubl

Optimized Strings for the Java HotSpot™ VM

A thesis submitted in partial satisfaction of
the requirements for the degree of

Master of Science
(Diplom-Ingenieur)

Supervised by

Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

Dipl.-Ing. Dr. Christian Wimmer

Institute for System Software
Johannes Kepler University Linz

Linz, June 2008

Abstract

Each string in Java consists of two separate objects. Metadata such as the length of the string are stored in the string object. For storing the characters, a separate character array is used. This separation results in increased memory usage and bad cache behavior because string operations must access both objects.

The optimization presented in this thesis merges the string object and the character array into a single object. Such an optimized string has a better cache behavior and requires less memory. The optimization is implemented for Sun Microsystems' Java HotSpot™ VM and is automatically applied at run time. The class `String` is optimized manually and all methods that allocate string objects are modified at run time. The modification at run time is performed once when loading a class and impacts the performance only negligibly.

Several benchmarks show a reduction of the memory usage and a performance increase. For the SPECjbb2005 benchmark, the performance increases by 8% and the memory usage is reduced by 3%. The performance of the DaCapo benchmarks improves by up to 14% (4% on average) and the memory usage is reduced by 5% on average. For the SPECjvm98 benchmarks, the performance improves by up to 62% (8% on average).

Kurzfassung

Jeder String in Java besteht aus zwei separaten Objekten. Metadaten wie die Länge des Strings werden im String Objekt gespeichert. Die Zeichen sind in einem eigenen Character Array gespeichert. Diese Aufteilung führt zu einem erhöhten Speicherbedarf und zu einem schlechten Cache Verhalten, da für String Operationen auf beide Objekte zugegriffen werden muss.

Die in dieser Diplomarbeit vorgestellte Optimierung verschmilzt das String Objekt und das Character Array zu einem Objekt. Dieser optimierte String hat ein besseres Cache Verhalten und benötigt weniger Speicher. Die Optimierung ist für Sun Microsystems' Java HotSpot™ VM implementiert und wird automatisch zur Laufzeit durchgeführt. Die Klasse `String` wird manuell optimiert und es werden alle Methoden, die String Objekte allokieren, zur Laufzeit modifiziert. Diese Modifikation zur Laufzeit erfolgt einmalig beim Laden einer Klasse und hat daher einen vernachlässigbaren Einfluss auf die Geschwindigkeit.

Mehrere Benchmarks zeigen eine Reduktion des Speicherbedarfs und eine Steigerung der Geschwindigkeit. Beim SPECjbb2005 Benchmark steigt die Geschwindigkeit um 8% und der Speicherbedarf wird um 3% reduziert. Die Geschwindgkeit der DaCapo Benchmarks steigt um bis zu 14% (durchschnittlich 4%) und der Speichbedarf reduziert sich um durchschnittlich 5%. Bei den SPECjvm98 Benchmarks verbessert sich die Geschwindigkeit um bis zu 64% (durchschnittlich 8%).

Contents

1 Introduction **1**
1.1 Development Steps . . . 2
1.2 Structure of the Thesis . . . 2

2 The Java HotSpot^{TM} Virtual Machine **4**
2.1 Overview . . . 4
2.2 Class File . . . 6
2.3 Interpreter . . . 8
2.4 Just-in-Time Compiler . . . 8
2.4.1 Server Compiler . . . 10
2.4.2 Client Compiler . . . 10
2.5 Memory Management . . . 11
2.5.1 Object Allocation . . . 12
2.5.2 Garbage Collector . . . 13
2.6 Object Header . . . 14
2.7 Java Native Interface . . . 16
2.8 Reflection . . . 17
2.9 Defensive Programming . . . 18
2.9.1 Assertions . . . 18
2.9.2 Impossible cases . . . 19
2.9.3 Self-Verification . . . 19

3 Java Strings **21**
3.1 Java String Class . . . 22
3.2 Optimized String Class . . . 23
3.3 Advantages and Disadvantages of the Optimization . . . 24
3.3.1 Advantages . . . 24
3.3.2 Disadvantages . . . 26

4 Implementation **28**
4.1 Removing the Field `offset` . . . 28
4.2 Removing the Field `hashcode` . . . 31

4.3 Character Access . . . 33
4.4 String Allocation . . . 35
4.5 Bytecode Rewriting . . . 37
4.6 Java Programming Language Compiler . . . 40
4.7 Heap Management Class Hierarchy . . . 41
4.8 Layout Helper . . . 43
4.9 Interpreter . . . 47
4.10 Just-in-Time Compiler . . . 48
4.11 Elimination of Explicit String Copying . . . 51

5 Evaluation **53**
5.1 SPECjbb2005 . . . 55
5.2 Elimination of Explicit String Copying for SPECjbb2005 . . . 59
5.3 DaCapo . . . 62
5.4 SPECjvm98 . . . 64

6 Related Work **67**

7 Conclusions **71**
7.1 Problems and Further Improvements . . . 71
7.2 Outlook . . . 72

List of Figures **74**

List of Listings **76**

Bibliography **77**

Acknowledgments

I want to thank all people who supported me during my studies and during the development of this thesis. Especially, thanks to my supervisor Christian Wimmer for his guidance, the numerous discussions, and valuable hints. Furthermore, thanks to him for running the benchmarks on his workstation to enrich the evaluation of the optimization. My thanks go to Hanspeter Mössenböck for his comments and for the funding.

I am most grateful to my parents, my brother Martin and my girlfriend Elisabeth for their long term support and encouragement. I want to express my gratitude to all my friends for their good company and the support over the years.

Chapter 1

Introduction

Strings are a common data structure and are used frequently in applications. Therefore, optimizing strings has a high potential to increase the performance. In this master's thesis, a new string optimization approach for Java is implemented and evaluated.

Each Java string stores metadata like the string's length and a reference to a character array. This corresponding character array is a separate object that stores the string characters. Due to this separation, it is possible that multiple string objects share the same character array. This sharing of character arrays is extended by the fields `offset` and `count`, which store the string's starting position within the character array and the number of characters that belong to the string. This allows character array sharing even if only a part of the character array is used. However, these fields increase the memory usage and have a negative impact on the performance.

Measurements with multiple benchmark suites yield that strings share their character array only in rare cases. Therefore, this thesis proposes an optimization that removes all unnecessary fields and merges the character array with the string object. This is possible because string objects are immutable and therefore the character array must not change during the lifetime. This voids the sharing of character arrays between string objects, but it offers other advantages. This optimization is implemented for Sun Microsystems' Java HotSpot™ VM, so that it is performed at run time behind the scenes and no actions on part of the programmer are required. In the VM, it is necessary to apply changes to the interpreter, the just-in-time-compiler, and to several other subsystems. The evaluation shows that the optimization increases the performance and reduces the memory usage.

An additional "optimized hash field" variant also removes the field `hashcode`. This field caches the string hash code after the first computation, so that the hash code must be

computed only once. The removal saves four bytes per string object but would preclude the caching of the hash code. Therefore, an approach was implemented that caches the hash code in the object header instead of the separate field `hashcode`.

1.1 Development Steps

This master's thesis was planned and implemented in multiple steps. The details for each step are presented in the subsequent chapters. In contrast to the listing below, the sequence of the chapters does not necessarily reflect the implementation order.

1. Analyzing the advantages and disadvantages of the optimization.
2. Removal of the string field `offset` within the Java source code and the Java HotSpot™ VM.
3. Introduction of new bytecodes for optimized string objects.
4. Modification of the Java programming language compiler (javac) for compiling the optimized string class.
5. Implementation of a bytecode rewriting heuristic.
6. Implementation of the new bytecodes in the interpreter.
7. Modification of internal data structures of the Java HotSpot™ VM.
8. Implementation of the new bytecodes in the just-in-time compiler.
9. Implementation of additional optimizations.
10. Implementation of the "optimized hash field" variant.
11. Benchmark execution and evaluation of the results.

1.2 Structure of the Thesis

Chapter 2 describes Java virtual machines in general, and focuses on Sun Microsystems' Java HotSpot™ VM. The most important components like the interpreter and the just-in-time compiler are discussed in detail, and general Java related topics like reflection or the Java Native Interface are covered. These parts are necessary to understand the implementation of the optimization described in Chapter 4.

The original and the optimized Java `String` classes are illustrated in Chapter 3. This chapter also presents the results of a detailed analysis of the advantages and the disadvantages of the optimization in comparison with original strings.

Chapter 4 is the main chapter of this thesis and describes the implementation details of the string optimization, like the bytecode transformation heuristic and the new bytecodes for the optimized string objects. Furthermore, this chapter also covers some non-successful approaches and difficulties.

The optimization is evaluated in Chapter 5. Multiple benchmarks were executed on two different machines. The performance gain and the reduction of memory usage are measured as two different aspects of the optimization. The benchmark results are diagrammed and discussed.

Chapter 6 presents related work to this master's thesis. The differences between the related work and the work done under the scope of this master's thesis are described.

The results of this master thesis are summarized in Chapter 7. Furthermore, problems with the current implementation as well as some ideas of further optimizations are stated.

Chapter 2

The Java HotSpot^{TM} Virtual Machine

For the platform independence of Java applications, a virtual machine (VM) is required. Java source code is compiled by a Java programming language compiler to machine-independent bytecodes that are executed by a Java virtual machine (JVM) [12]. The specification of a JVM is given in an abstract way [21], so that different implementations and optimizations are possible. Today, multiple companies provide or sell their JVM implementations.

Sun Microsystem's Java HotSpot^{TM} VM is written in the programming language C++ and is platform dependent. An interpreter starts with the bytecode execution, but selected methods can be compiled to platform dependent machine code to improve the performance. To further speed up the execution, multiple optimizations are applied while compiling bytecode to machine code.

2.1 Overview

A JVM is a program that supports the Java instructions set and executes platform independent Java bytecodes. The Java instructions set specifies 205 different bytecodes. The maximum number is limited to 256 because only one byte is used for encoding. Within a JVM, bytecodes that are not contained in the official instruction set can be used for the implementation of optimizations.

In principle, a JVM is a stack machine and the bytecodes pop or push operands to or from the operand stack. A Java method is a sequence of bytecodes and their operands. For the execution of a method, the method can be interpreted or compiled as presented in Figure 2.1. Interpreting a method is slow and therefore a method can be compiled to machine code. Because the compilation is performed at run time, the necessary time adds to the total execution time of the application. To determine for which method a compilation is beneficial, each invocation of a method is recorded by increasing its invocation counter. If the invocation counter reaches a certain threshold, the method is compiled. This uses the fact that only a small percentage of the code is executed frequently and therefore contributes the main part to the total execution time.

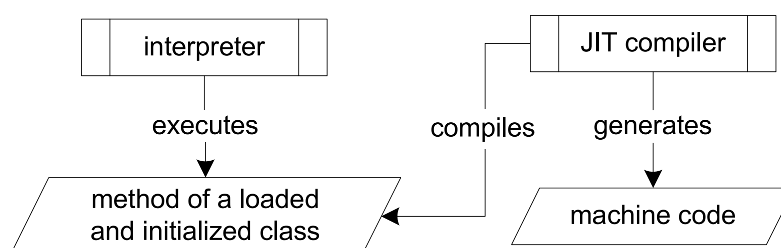


Figure 2.1: Execution of a method

Before a method can be executed, its class must be loaded by the JVM. For this purpose, a class loader dynamically loads the required classes. The bootstrap class loader is directly built into the JVM. User defined class loaders can be implemented to extend the class loading behavior. It is for example possible to implement a class loader that retrieves the class file over a network or reads it from an encrypted file. All class loaders except the bootstrap class loader reference a parent class loader. If a class loader should load a class, it first delegates the class loading to its parent class loader. Only if the parent class loader fails to load the class, the class loader tries to load it itself. A loaded class is identified by its fully qualified name and its class loader. Therefore, two classes with the same fully qualified name, but with different class loaders are treated as different classes. Figure 2.2 illustrates the basic procedure of loading a class into the JVM. The procedure involves the following steps:

- A Java programming language compiler like Sun Microsystems' javac is used to compile the Java source code to platform independent bytecodes. A class file is generated that contains the bytecodes and additional metadata.
- The class loader retrieves the class file, parses it and verifies its structure. Then, an internal representation of the classes, interfaces, methods, fields, and constants is generated.

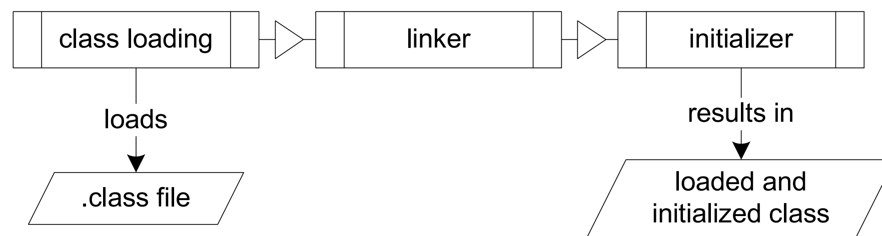


Figure 2.2: Class loading

- Upon the first execution, the loaded class is linked. This involves a bytecode verification process that ensures that no broken or invalid bytecode is executed. Then, the symbolic references in the bytecode can be resolved to actual classes, interfaces, fields, or methods. This might involve the consecutive loading of other classes or interfaces.
- In the last step, the loaded class is initialized: the static initializer is executed, so that values are assigned to static fields. After this step, bytecodes that allocate or access an instance of this class can be executed.

In contrast to other object oriented languages, Java allocates all objects on the heap. Only as an optimization for objects that do not escape a method or thread, stack allocation might be used [17]. However, this behavior is an optimization optionally performed by the Java HotSpot^{TM} VM and cannot be influenced by the programmer. For the deallocation of objects, a generational garbage collector [16] is used. This garbage collector runs periodically and deallocates objects that are no longer referenced.

2.2 Class File

The Java programming language compiler compiles Java source code to a class file [12]. The file encapsulates various informations that are used for the execution [21]:

- **Basic information:** The class name, the super class, implemented interfaces, and access flags are stored. The access flags correspond to the access flags specified in the Java source code. When a class is loaded, this information is used to determine if other classes must be loaded consecutively.
- **Constant pool:** This data structure is similar to a symbol table, where string constants, the names of classes, interfaces, and fields as well as other constants are stored. If a class uses the same constant multiple times, it is only stored once. All

code positions that need this constant reference it through an index into the constant pool.

- **Fields:** The name, the access flags, the field type, and optional annotations are stored for every field. When a field is accessed, it is queried if the corresponding class has a field with the specified name and if the flags allow the access.
- **Methods:** For each method, the name, the access flags, the signature, and optional annotations are stored. Furthermore, for all methods that are not declared as native or abstract, the Java bytecodes are present.

The information in the constant pool is highly distributed, as illustrated in Figure 2.3. It shows the constant pool entries for the constructors `String(char[] ch)` and `String(byte[] b)`. The entries 0 to 3 store string constants that are referenced by the other entries, and entry 4 stores the class `String`. The method descriptors are stored in the entries 5 and 6 and reference the same name, but use different signatures. The entries 7 and 8 store the fully qualified methods that share the same class but reference different methods. These two entries can be referenced by an `invokespecial` bytecode to invoke one of the constructors.

0	"String"	
1	"<init>"	
2	"(C[)V"	void … (char[] ch)
3	"(B[)V"	void … (byte[] b)
4	Name: #0	class String
5	Name: #1 Sig: #2	<init>(char[] ch)
6	Name: #1 Sig: #3	<init>(byte[] b)
7	Class: #4 Meth: #5	String.<init>(char[] ch)
8	Class: #4 Meth: #6	String.<init>(byte[] b)

Figure 2.3: Constant pool example

When a class file is loaded, the run time constant pool is created. This is the JVM internal representation that holds the same information as the class file, but can be more efficiently accessed by the interpreter and the just-in-time compiler. If the interpreter starts the execution of a method, the run time constant pool is used to retrieve the specific method.

2.3 Interpreter

After the startup of the Java HotSpot^{TM} VM, all methods are interpreted per default. The interpreter has a machine dependent code template for each bytecode. These templates are translated to machine code at the startup. If a method is interpreted multiple times, it is still nearly as slow as on its first invocation. This is the case because the resulting code from the first execution is not stored anywhere. During the bytecode execution, the interpreter collects run time information that is used to trigger the just-in-time compilation. Each time a method is invoked, its invocation counter is increased. If this counter reaches a certain threshold, the method is compiled to optimized machine code. Without exception handling, the interpreter repeats the steps shown in Listing 2.1 [21]. The next bytecode and its operands are loaded, so that the appropriate code template can be executed.

```
instruction = nextInstruction();

while(instruction != null) {
  if(instruction.hasOperands()) instruction.fetchOperands();
  instruction.execute();
  instruction = nextInstruction();
}
```

Listing 2.1: Interpreter loop

2.4 Just-in-Time Compiler

In a traditional compiler, the source code is compiled to machine code before the execution. The whole program is compiled once, so that compilation is not time critical. In Java, just-in-time (JIT) compilation is used to speed up the execution of an application [2]. In contrast to a traditional compiler, the compilation is performed during the execution, so that the compilation time adds to the total execution time. Therefore, it is necessary that the compilation is as fast as possible. Time consuming optimizations are often omitted. To reduce the total compilation time further, only selected methods are compiled. For deciding which methods should be compiled, the interpreter gathers run time information like the number of invocations for each method. If a specific invocation threshold is reached, the method is compiled.

Machine code is generated as a result of the compilation. This code is stored for the remaining execution time and replaces the original method, so that on subsequent invocations

the machine code is executed. JIT compilers have the disadvantage that the compilation time is critical and that therefore fewer optimizations can be applied. However, a JIT compiler can make use of processor and platform dependent optimizations that cannot be applied to conventionally compiled programs because of backwards compatibility.

Furthermore, the JIT compiler can perform optimistic optimizations that might have to be deoptimized later during the execution [14]. For the Java HotSpot^{TM} VM, method inlining is an important optimistically applied optimization. If a method is not overridden by any loaded subclass, it is a candidate for method inlining. Method inlining replaces the invocation of a method with the method's code, i.e. it copies the invoked method's code to the position of the invocation. This removes the overhead of the method invocation and enlarges the method's body to increase the opportunities of other optimizations. Without deoptimization, method inlining would be far more difficult in Java because all class methods are virtual per definition. If a subclass that overrides an inlined method is loaded during the execution, the inlining is voided because the inlined base class method would always be executed instead of the overridden subclass method. Therefore, the affected method must be recompiled. Deoptimization is only necessary if the affected method is currently being executed. In this case, the Java HotSpot^{TM} VM falls back from the compiled machine code to the interpreter.

Because the performance of a JIT compiled application depends on the number of compiled methods, it changes over time. Two metrics are used to characterize the performance:

- **Startup time:** When the execution of an application is started, all methods are interpreted. This is slow and therefore the overall performance is low. When the first methods are compiled, the compilation adds a significant amount of additional workload that precludes a performance gain. The longer the JIT compilation takes, the higher is the startup time. A high startup time can be afforded on servers, where mainly long running applications are executed and where a high peak performance is to be achieved. For client applications, the startup time must be as low as possible not to degrade the user experience.
- **Peak performance:** After the startup time, the most important methods are compiled and the application's performance converges towards its maximum. The peak performance is determined by the quality of the optimizations applied during the JIT compilation.

Therefore, Sun Microsystems delivers two different JIT compilers: the server compiler and the client compiler. The optimization presented in this master's thesis is only implemented

for the client compiler and therefore the following description of the two JIT compilers focuses on the client compiler.

2.4.1 Server Compiler

The server compiler performs time consuming optimizations in order to generate as good as possible optimized machine code [22]. On a server, mostly long running applications are executed, so that the longer compilation time is only a small overhead in comparison to the total execution time. Furthermore, the compilation can be efficiently performed in the background if multiple processors are available, i.e. one of the processors can be used only for compilation. Like a traditional compiler, the server compiler uses the following phases for the compilation: parsing, machine-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 machine independent IR to machine dependent instructions. These machine dependent instructions 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 follows that 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.

2.4.2 Client Compiler

The client compiler is optimized for compilation speed and a small footprint [13, 18]. Therefore, all time consuming optimizations are omitted. With this strategy, an application's startup time is low and the generated code is still reasonably well optimized, so that a good peak performance is achieved. The compilation of a method consist of 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 [9] and suitable for global optimizations. A C++ class hierarchy models the different high level instructions, where each instruction represents both the computation

and the result. Therefore, instructions might have other instructions as arguments. 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. All instructions belong to a basic block, until a jump or a jump target is encountered. Such an instruction terminates the basic block and starts a new one. 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 use of specific machine registers for specific instructions, can be modeled directly in the LIR, which simplifies the code generation. Linear scan register allocation [32] is used to map all virtual registers to physical ones.

Code generation finishes the compilation of a method. This process is rather simple because register allocation was already performed previously. Therefore it is only necessary to translate each LIR instruction to the machine dependent instructions. Uncommon cases like throwing an exception or a necessary garbage collection are handled as separate cases, which are emitted at the method's end. Additionally to the machine code, other for the execution required information is generated.

Within the Java HotSpot^{TM} VM, it is possible to declare specific methods as intrinsic. If such an intrinsic method is to be compiled or inlined, a handcrafted sequence of instructions or a fixed piece of machine code is used. This provides the possibility of manually optimizing specific methods.

2.5 Memory Management

In many programming languages, the programmer explicitly deallocates objects. This has the advantage that the moment of the deallocation is known and that the programmer can free an object immediately after it is ensured that it is no longer needed. On the other hand, explicit memory management is complex and might lead to a variety of errors. In Java, automatic memory management is performed so that the programmer does not need to deallocate objects manually [28].

2.5.1 Object Allocation

The allocation and initialization of an object is performed by two different bytecodes in Java. Figure 2.4 illustrates the bytecodes for allocating objects and arrays [21]. For the allocation of an object, its size must be known. The size of an class instance is statically known because of its base class and its fields. The bytecode `new`, which is used for the allocation of class instances, has as its parameter a constant pool entry to a class descriptor object. Upon the execution, the constant pool entry is resolved and the referenced class descriptor object is used to determine the instance size. After retrieving the size, the instance is allocated.

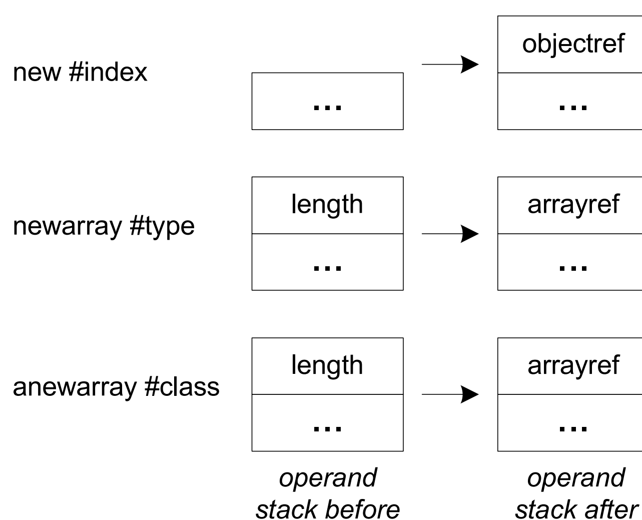


Figure 2.4: Bytecodes used for allocation

Arrays are the only data structure in Java where the size is not known statically. Therefore, the instructions `newarray` and `anewarray` are used for allocating type or object arrays. The only statically known fact about arrays is the element type, which is one of the operands of the instructions. As a second operand, the desired length is required on top of the operand stack. With this information, the total array size is calculated: the array length is multiplied with the size of the element type and the array's header size is added to the result. After this calculation, the allocation is performed.

All Java objects are aligned to addresses that are divisible by 8 bytes, both on 32-bit architectures and on 64-bit architectures. If an object's size is not a multiple of 8 bytes, space is wasted. Figure 2.5 shows this on a string object with a length of zero characters. The character array has a size of 12 bytes, which is not divisible by 8. Therefore, 4 bytes

are wasted. The string object has a size of 24 bytes, which is divisible by 8, so that no space is wasted.

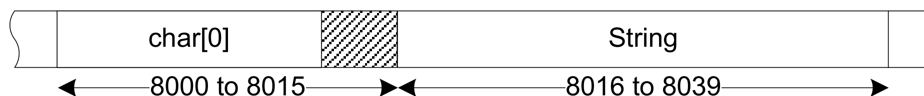


Figure 2.5: Heap with a string object and 8 byte alignment

2.5.2 Garbage Collector

Instead of deallocating objects manually, a generational garbage collector is used in the Java HotSpot^{TM} VM [16]. The garbage collector runs periodically and collects all objects that are no longer referenced. For this task, it is necessary that the GC has exact information about all objects. Especially the positions of all pointers to other objects must be known. This information is used to determine if an object is still referenced by others, or if it is already unreachable. A garbage collection can occur only on specific positions, so called safepoints. Method calls and operations that may throw an exception are examples for instructions where a safepoint is created. If the default garbage collector is used, the whole program execution is stopped each time a garbage collection occurs. The heap is divided into three generations for a more efficient heap management:

- **Young generation:** Most objects only have a very short lifetime and therefore a separate generation for newly allocated objects is used that is managed by a stop-and-copy garbage collector. The young generation is usually small in comparison to the total heap size. Many fast garbage collections (so called minor garbage collections) occur for this generation. If an object has survived several garbage collections, it is promoted to the tenured generation.
- **Tenured generation:** This generation is managed by a mark-and-compact garbage collector and occupies the remaining part of the heap. Only few garbage collections (so called full garbage collections) occur for this generation, but those need lots of time because of the large size. Long garbage collections might degrade the user experience because the application does not respond while the garbage collector is running.
- **Permanent generation:** This area usually has a fixed size and is reserved for internal data structures that are mostly needed during the whole execution time of the application. No longer required objects in this generation might be deallocated on a full garbage collection.

Some garbage collectors try to fulfill certain predefined goals like minimizing the duration of a garbage collection. This can result in a higher number of total garbage collections and therefore a higher total garbage collection time. However, it could be more feasible for applications that require a high response time.

The garbage collector uses per default only one processor. This causes a major performance degradation on server machines with a large physical memory and multiple processors because the execution of the whole application must be stopped for the time of the garbage collection. That is why other garbage collectors are provided by the Java HotSpot^{TM} VM. A parallel garbage collector reduces the garbage collection overhead by using multiple processors. Alternatively, a concurrent mark-and-sweep algorithm allows the application to continue its execution during the garbage collection. For this, some processors are permanently assigned to the garbage collection. However, every garbage collector has some disadvantages, so it depends on the scenario which one is the best.

2.6 Object Header

Each Java object has a header of two machine words, i.e. 8 bytes on 32-bit architectures and 16 bytes on 64-bit architectures [23]. Figure 2.6 illustrates the object layout for a character array: the object header is followed by the array length and the actual characters.

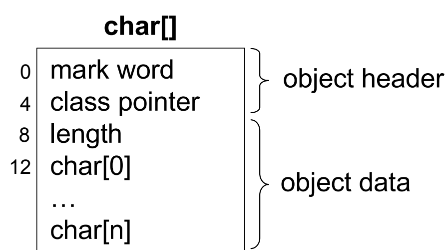


Figure 2.6: Java object header on 32-bit architectures

The first machine word, the so called *mark word*, stores the identity hash code and the object's 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()`. Although the identity hash code is preferably unique for each object, this is not guaranteed. Because the identity hash code must not change for a specific object during its lifetime and because it does not depend on the object's content, it cannot be recomputed and therefore must be cached. If the object is unlocked, the identity hash code can be stored in the

mark word. 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. Based on the status information stored in the *mark word*, an object is in one of five states shown in Figure 2.7:

- **Unlocked:** An object in this state can be accessed by any thread without any restrictions. Only in this state, the identity hash code can be stored in the *mark word*.

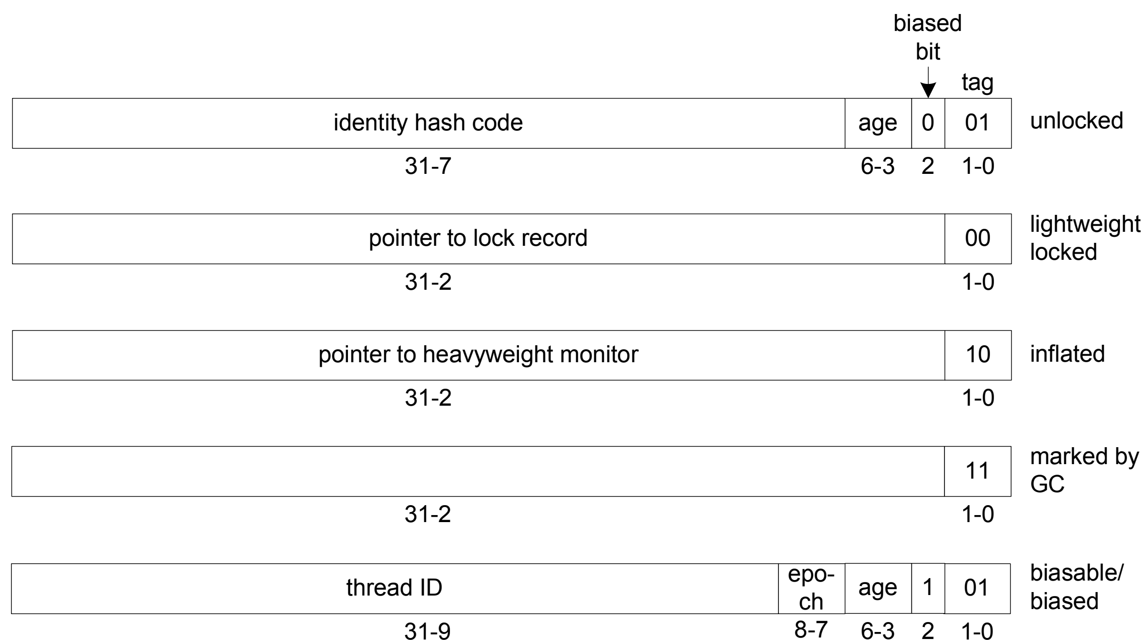


Figure 2.7: Possible bit patterns for the *mark word*

- **Lightweight locked:** When the bytecode `monitorenter` is used to lock an object, it is lightweight locked. A lock record is allocated where metadata and the content of the *mark word* (e.g. the identity hash code) are stored. When the object is unlocked, the saved *mark word* is restored.
- **Inflated:** If thread `B` tries to acquire a lock for an object that is already locked by thread `A`, the object's status is changed to inflated. When thread `A` unlocks the object, it recognizes that another thread is waiting for the lock and notifies this thread.
- **Marked:** This state is only allowed during garbage collection and is used by the garbage collector to identify which objects are still reachable. The garbage collector

follows all pointers and sets the reachable objects to the marked state. After the mark phase, all objects that are not marked are deallocated.

- **Biasable/Biased:** Biased locking is an optimization to reduce the number of atomic operations necessary for locking an object. Per default, biased locking is enabled and therefore all objects are allocated with the biasable pattern in the *mark word*. As long as the stored thread ID is zero, the object is not yet biased to any thread. If the identity hash code is computed for an object in this state, the object must be unbiased so that the state can be changed to unlocked.

The second machine word stores a pointer to a class descriptor object. This class descriptor is unique for all objects of a specific class and holds the metadata and the method table of the class. Furthermore, it is for example used for the object size computation and for reflection.

2.7 Java Native Interface

The Java Native Interface (JNI) is a standardized interface that specifies the interoperability between Java and platform dependent binaries [20]. It enables Java application to invoke platform dependent methods that are written in other programming languages such as C++. Furthermore, it allows platform dependent code to call Java methods or to run a complete JVM. It is mainly used to implement platform dependent features or time critical operations that cannot be implemented completely in Java. The JNI provides lots of operations such as allocating Java objects or calling Java methods. All these operations are directly implemented in C++ within the Java HotSpot^{TM} VM.

To call a specific method in a platform dependent library, the library must be loaded via the method `System.loadLibrary()`. A specific naming scheme is used to identify JNI methods in the set of all exported methods of the library. The code example in Listing 2.2 illustrates the usage of the JNI. The static constructor of the Java class `Calculator` loads the library that implements the native method `Calculator.power()`. The C++ implementation of the method is bound to the native Java method, so that it can be invoked by any Java code.

```
public class Calculator {
  static { System.loadLibrary("calculate"); }

  public static native int power(int base, int exp);
}
```

(a) a class that uses a JNI method

```
jint Java_Calculator_power__II(JNIEnv *env, jobject obj,
   jint base, jint exp) {

   jint result = ...;
   return result;
}
```

(b) C++ code for a JNI method

Listing 2.2: Example usage of the Java Native Interface

2.8 Reflection

Reflection is used to determine information about Java objects and their parts at run time [5]. Furthermore, the fields of an object can be accessed and modified, and its methods can be invoked. Typical examples for the use of reflection are class browsers, debuggers or tools for automatic testing. The example presented in Listing 2.3 retrieves the `Class` object for strings and prints all available constructors to the console.

```
public class Reflection {
  public void printConstructorNames() {
    try {
      Class<?> clazz = Class.forName("java.lang.String");
      Constructor<?>[] ctors = clazz.getConstructors();

      for(Constructor<?> c: ctors) {
        System.out.println(c.toGenericString());
      }
    } catch(SecurityException e) {
      e.printStackTrace();
    } catch(ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Listing 2.3: Example usage of reflection

2.9 Defensive Programming

Defensive programming can greatly help to find errors [11]. Especially in a project as big as the Java HotSpot^{TM} VM, all precautions to prevent or to find errors more efficiently are highly effective. If a programmer changes some details in his implementation, he maybe is not aware of the effects on the other parts of the JVM. Therefore, it is useful that each programmer implements additional checks to validate the preconditions, postconditions, and invariants. Most checks are only enabled in the debug build of the JVM and do not reduce the performance of the product version. After finishing changes on the source code, a debug build is created and used for testing. The Java HotSpot^{TM} VM uses several techniques for defensive programming that are presented in the following sections.

2.9.1 Assertions

Assertions are the basic technique for defensive programming. They check a condition at a specific location in the implementation. If the condition does not hold, an error message is printed and the Java HotSpot^{TM} VM exits. An example for such an assertion is shown in Listing 2.4: a programmer assumes a heap-word size of four bytes and implements an optimization based on this.

```
int size_in_heap_words = ...;

assert(HeapWordSize == 4, "for optimizing the next line");
int size_in_bytes = size_in_heap_words << 2;
```

Listing 2.4: Example usage of assertions

If the heap-word size changes, the optimization produces a wrong result. With the assertion in place, the invalidated assumption is easily detected and the optimized code can be changed. Without, this might lead to an error within the garbage collector, which relies on the correct object size. The mistake is propagated to a completely different part of the JVM, where multiple possible error sources exist and where the costs for finding the mistake are much higher.

2.9.2 Impossible cases

Another way of securing the code is to explicitly handle all possible cases for the statements `switch` and `if`. If it is assumed that only two cases are possible, a programmer often uses an `if-else` construct, where the `else` branch does not have any associated condition. However, this assumption might not hold any longer if new features are added. All unknown cases are then handled by the `else` branch, which most likely produces a wrong result.

Therefore, all known cases should be explicitly checked, so that the `else` branch can be declared as an impossible case, as shown in Listing 2.5. If the `else` branch is still taken, an error is reported and the JVM exits. The same applies to the `default` case of `switch` statements. Although this technique slightly increases the work for the implementer, it is especially useful if new features are added to an existing project.

```
int bytecode = ...;

if(bytecode == _new) {
  alloc_object();
} else if(bytecode == _newarray) {
  alloc_array();
} else {
  should_not_reach_here();
}

switch(bytecode) {
  case _new:      alloc_object(); break;
  case _newarray: alloc_array(); break;
  default:        should_not_reach_here(); break;
}
```

Listing 2.5: Example for impossible cases

2.9.3 Self-Verification

Self-verification uses verification methods to compare the actual behavior of a program part with the specified one. This can for example be done by inspecting internal data structures. Because these verification methods can heavily reduce the performance, they are not executed in the product version. An example for self-verification is shown in Listing 2.6: after inserting a string into a hashtable, the string is immediately retrieved to ensure that it is stored properly. In comparison to the other techniques, self-verification is more complex and more expensive to implement.

```
Hashtable::put(int key, jchar* value) {
  ...    // store the value

  #ifdef DEBUG
    jchar* in_table = get(key);
    assert(value == in_table, "value must be in table now");
  #endif
}
```

Listing 2.6: Example for self-verification

If a performance critical method of an application is at first naively implemented and optimized later on, it is likely that the optimized version contains a bug because optimizations are more difficult to implement. Therefore, it is useful to keep the unoptimized method and to compare the results of both methods in a debug build. This is nearly no effort to implement and can still help to detect bugs.

Chapter 3

Java Strings

Java allows to declare strings directly at the language level, and it provides the operator "+" for string concatenation [12]. However, all string operations are compiled to method calls of the classes `String` and `StringBuilder`. Every string stores Java characters that are defined as Unicode characters with a size of two bytes. In contrast to strings in programming languages like C, Java strings are not null-terminated because each string stores its length. Additionally to the basic data structure, the class `String` contributes algorithms for string manipulation. However, methods that change a string's content always create a new string object because strings are immutable, i.e. the characters of a string must not change after initialization. Therefore, strings are not suitable if lots of modifying operations are executed.

For operations, where the string characters change often, the classes `StringBuffer` or `StringBuilder` should be used. These data structures can be seen complementary to strings and hold a mutable character array that grows dynamically. Different optimizations can be applied to the data structures, and all are efficient if used correctly. However, the programmer must decide which one is suitable in which situation.

In this chapter, the original and the optimized Java `String` classes are described. Furthermore, this chapter presents the results of a detailed analysis of the advantages and the disadvantages of the optimization.

3.1 Java String Class

Figure 3.1 illustrates that a string consists of two objects: metadata like the string length are stored in the actual string object and the string characters are stored in a separate character array. Because of this separation, multiple string objects can share the same character array. This sharing of character arrays is extended by the fields `count` and `offset` that store the string's length and the starting position within the character array. Therefore, a string does not need to use the full character array. Methods such as `String.substring()` make use of this: instead of copying the selected characters to a new character array, a new string object is allocated that references the same character array. Only the starting offset and the length of this newly allocated string object are set. This technique reduces the number of character arrays with the same content, which is positive for the memory usage and the performance. The hash code of a string object is computed from the string characters. To avoid recomputations, the hash code is cached in the field `hashcode`. A string object has a minimum size of 24 bytes without its character array, and 36 bytes with a character array that contains zero characters.

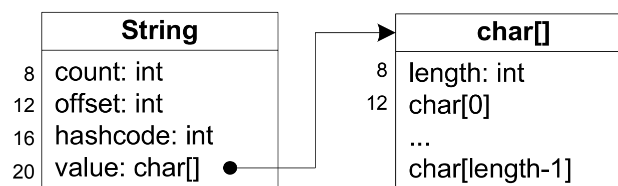


Figure 3.1: Layout of Java strings

The field `offset` increases the memory usage and it must be loaded when a string character is accessed to determine the string's starting position. The actual array is then computed by adding the offset to the character's index. The field access and the computation are an overhead with a negative impact on the performance. Furthermore, it is possible that a string uses a far too large character array. The garbage collector is not aware of this and cannot free any unnecessary used memory because it determines that the character array is still referenced by a string object. If a string object uses the whole character array, the field `count` wastes memory because it is a duplication of the character array length. Another disadvantage of original strings is that each time a string object is accessed, two bounds checks are performed: 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 executed when the character array is accessed. This duplication reduces the performance.

A modified class `String` was used to measure some facts about character array sharing between string objects for the benchmarks SPECjbb2005 [27], SPECjvm98 [26], and DaCapo [3]. The percentage of strings that do not use their full character array is 0.05% for the SPECjbb2005 benchmark, 5% for the SPECjvm98 benchmarks, and 14% for the DaCapo benchmarks. These results indicate that it is uncommon that strings use only a part of their character array. However, a string also shares its character array when the string is explicitly copied. Of all string allocations, 19% are explicit string copies for the SPECjbb2005 benchmark and 4% for the SPECjvm98 benchmark. The DaCapo benchmark uses hardly any explicit string copies.

3.2 Optimized String Class

Because it is not common that a string shares its character array, it is beneficial to remove all unnecessary fields and to merge the character array with the string object. Merging the character array with the string object is possible because string objects are immutable and the character array is declared as private. This has the disadvantage that it precludes the sharing of character arrays between string objects. However, it offers several other advantages like the reduction of memory usage and the elimination of field accesses, as described in the next section.

Figure 3.2 shows string objects for the two implemented variants of the optimization. The basic optimization, shown in Figure 3.2 (a), merges the character array with the string object. Furthermore, it removes the string field `offset` and the character array field `length` to reduce the memory usage.

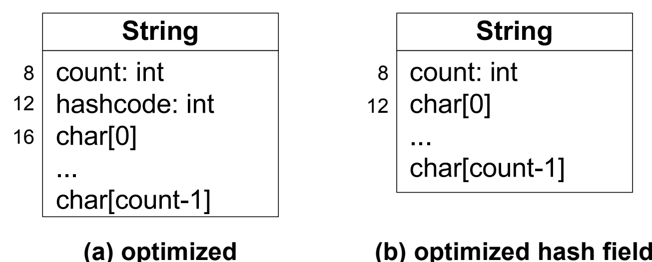


Figure 3.2: Layout of optimized strings

The optimization shown in Figure 3.2 (b) also removes the field `hashcode` and saves another four bytes per string object. The calculated hash code is cached in the *mark word* of the object header, where normally the identity hash code is stored. Therefore, the method

`String.hashCode()` returns the same value as the method `System.identityHashCode()` for such optimized string objects. On 32-bit architectures, the hash code is truncated to 25 bits because the *mark word* is not large enough to hold the value without truncation. The hash code algorithm for strings is documented in the Javadoc of the class `String`, so that this optimization violates the specification on 32-bit architectures. On 64-bit architectures, the *mark word* is large enough to cache the hash code without truncation, so that this optimization complies with the specification.

3.3 Advantages and Disadvantages of the Optimization

The presented string optimization tries to fulfill two competing goals: the performance of string objects should be increased and the memory usage should be decreased. Because the sharing of character arrays is precluded by the optimization, some string methods have a decreased performance and an increased memory usage. Therefore, all positive and negative effects on the memory usage and the performance are discussed in the next two sections.

3.3.1 Advantages

- **Fewer field accesses:** Original string objects load the field `offset` for nearly all operations. To access a specific string character, the field `offset` must be added to the character's index to determine the actual access position. With the removal of the field `offset`, the field access and the computation of the access position are no longer necessary.
- **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.
- **Better cache behavior:** An original string consists of two separate objects that might be spread over the heap by the garbage collector. Figure 3.3 shows a possible memory layout before and after garbage collection. For accessing the string characters, both parts must be accessed, which leads to a bad cache behavior. Optimized strings have a good cache behavior because they consist of only one object that might fit into a cache line.
- **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.

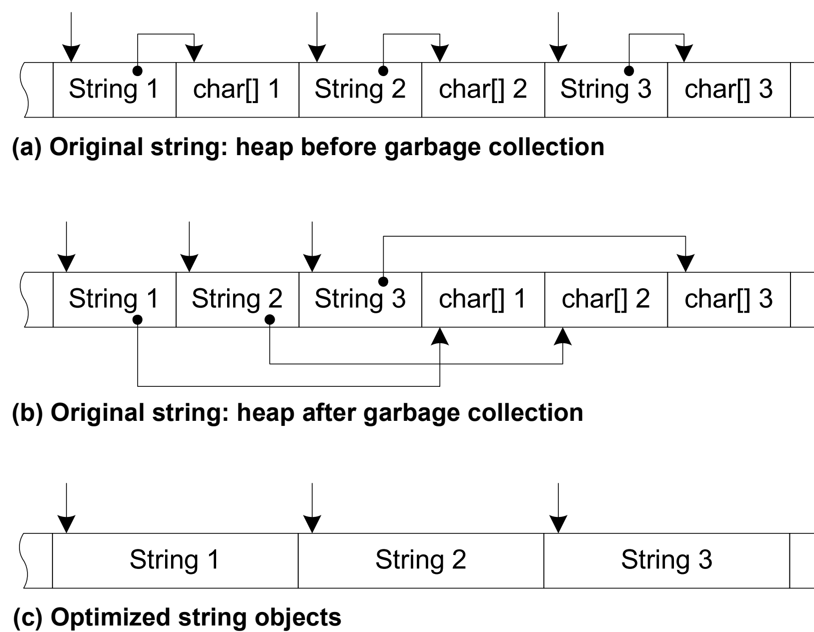


Figure 3.3: Cache behavior of strings

A second implicit bounds check is performed when the character array is accessed. This duplication does no longer exist for optimized strings.

- **Reduced memory usage:** An original string has a minimum size of 36 bytes (including the character array and ignoring the possibility of character array sharing). The minimum size of an optimized string object is 12 or 16 bytes, depending on the removal of the field `hashcode`. Therefore, up to 24 bytes are saved per string object.

  These saved bytes result in a percentage of reduced memory usage as shown in Figure 3.4. Because the string characters are not optimized in any way, the percentage of reduced memory depends on the string length. The figure illustrates the maximum savings and ignores the possibility of character array sharing. It also contains the mean string length for the various benchmark suites. For strings with a length of 10 characters, the memory usage is reduced by up to 44%. This results in fewer garbage collections, which affects the performance positively.

- **Faster allocation:** When an original string object and its character array are allocated, both are fully 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 might 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 used memory because it can only determine that the

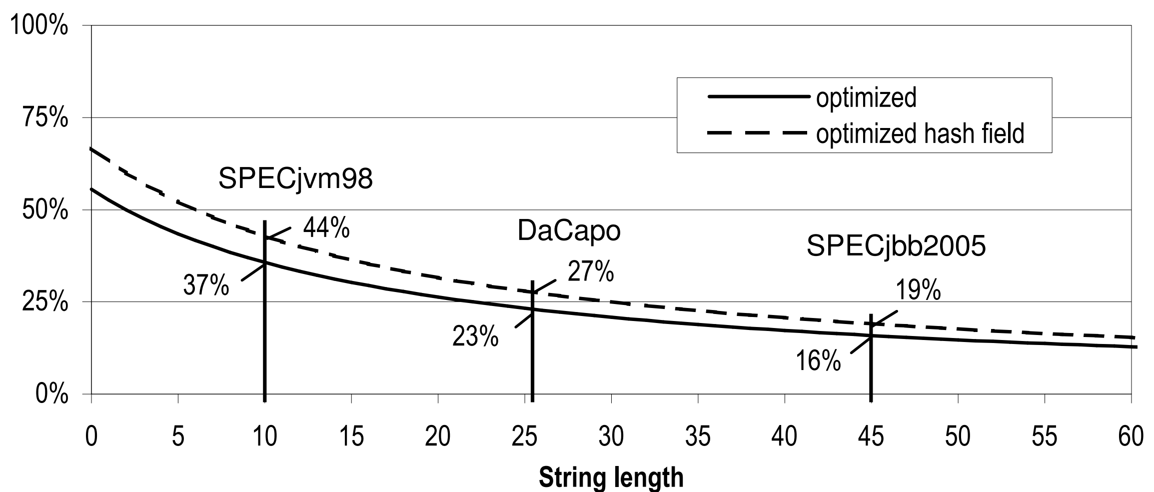


Figure 3.4: Reduction of memory usage for the optimized string class (higher is better)

character array is still referenced by a string object. Optimized strings use exactly the minimum necessary amount of memory for storing the characters.

3.3.2 Disadvantages

- **Rewriting:** Each method that allocates string objects needs some bytecode rewriting at class loading. This rewriting process takes additional time but has to be done only once. Therefore, it adds a negligible overhead compared to the total execution time. This is the case for both client and server applications.
- **Temporary character arrays:** For some methods of the class `String`, the internal character array is passed to a helper method of another class. This is not possible for optimized strings because the characters are not stored in a real character array. Therefore, a temporary character array is allocated to which all string characters are copied. This character array is then passed to such methods. The copying reduces the performance and increases the memory usage. This could be optimized further.
- **Voided character array sharing:** Merging the character array with the string object precludes character array sharing. Several methods that profited from character array sharing are therefore degraded in their performance and memory usage. For example, the method `String.substring()` only adjusts the fields `offset` and `count` for original strings. No characters must be copied.

With the optimization, it is necessary to create a new string object where the affected characters are copied to, as illustrated in Figure 3.5. Therefore, the runtime complexity of the method `String.substring()` is degraded from O(1) to O(n).

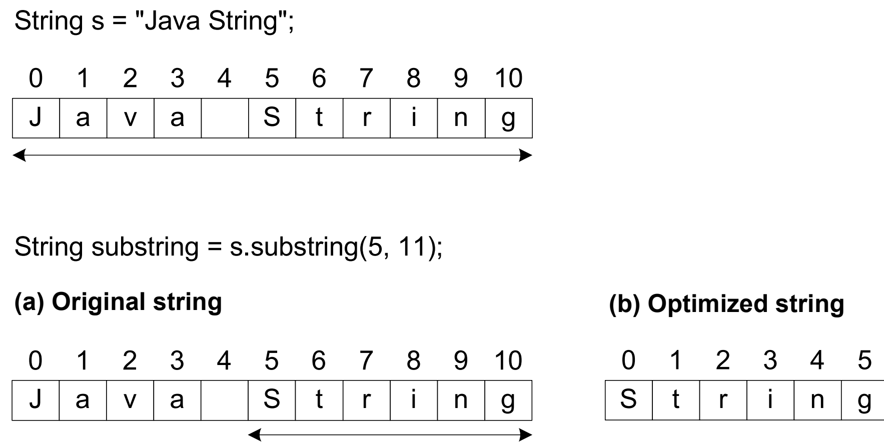


Figure 3.5: Method `String.substring()`

Chapter 4

Implementation

This chapter describes the implementation details of the optimization. The removal of the field `offset` is the first step to slim down string objects. For merging the character array with the string object, three new bytecodes are introduced. Because these new bytecodes are only used within the class `String`, the Java programming language compiler (javac) is modified to create the optimized string class file. Otherwise, the transformation to the optimized class would have to be done at runtime, which is less flexible and more effort to implement. To support the three new bytecodes, they are implemented in the JIT compiler and the interpreter. At runtime, all methods that allocate string objects must be modified because the optimization requires changes to the allocation of string objects. Additionally to the details of the implementation, some non-successful approaches and particular difficulties are described in this chapter.

The optimization is implemented for the early access version b24 of Sun Microsystems' Java HotSpot™ VM that is part of the upcoming JDK 7 [29]. Because platform dependent code is necessary within the interpreter and the just-in-time compiler, the optimization is currently only implemented for the IA-32 architecture.

4.1 Removing the Field `offset`

Optimized strings merge the character array with the string object, which precludes character array sharing. For original string objects, the field `offset` extends the possibility of character array sharing and can be removed for optimized strings. The Java source code of the class `String` is modified for this. During these modifications, the following three typical cases occur. All examples in this section represent the state after the removal of

the field `offset` and require further changes when the character array is merged with the string object.

- Most of the time, accesses to the field `offset` can be deleted or replaced by the constant 0 without causing any side effects. This has a positive impact on the performance because fewer field accesses are necessary. The method `String.charAt()`, shown in Listing 4.1, is an example for this case.

```
public char charAt(int index) {
  ... // bounds check
  return value[index + offset];
}
```

(a) original

```
public char charAt(int index) {
  ... // bounds check
  return value[index];
}
```

(b) removed field `offset`

Listing 4.1: Method `String.charAt()`

- If the field `offset` is used to implement an optimization using character array sharing, more effort is necessary because this kind of optimization is no longer possible. Optimized strings must copy all characters to a new string object, which costs some performance. This case occurs for example in the method `String.substring()`, as show in Listing 4.2. For original strings, a new string object, which references the same character array, is allocated and only the fields `offset` and `count` are set accordingly.

```
public String substring(int beginIndex, int endIndex) {
  ... // bounds check
  return new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

(a) original

```
public String substring(int beginIndex, int endIndex) {
  ... // bounds check
  int newCount = endIndex - beginIndex;
  char[] newValue = new char[newCount];
  System.arraycopy(value, offset, newValue, 0, newCount);
  return new String(newValue);
}
```

(b) removed field `offset`

Listing 4.2: Method `String.substring()`

- In a few cases, the string-internal character array is passed to a helper method of another class. The characters of optimized strings are no longer stored in a real character array that could be passed to such a method. Therefore, it would be necessary to pass the string object itself. For this, the receiving method would have to be overloaded to allow a string object instead of a character array as an argument.

  The decision was made to keep the number of changes to a minimum and therefore no methods outside of the class `String` are added or modified. Instead of this, the string characters are copied to a temporary character array, which is passed to such methods. This costs some performance and increases the memory usage but could be optimized further. The method `String.getBytes()` shown in Listing 4.3 serves as an example for this case.

```
public byte[] getBytes(String charsetName) {
  if(charsetName == null) throw new NullPointerException();
  return StringCoding.encode(charsetName, value, offset, count);
}
```

(a) original

```
public byte[] getBytes(String charsetName) {
  if(charsetName == null) throw new NullPointerException();
  char[] val = toCharArray();
  return StringCoding.encode(charsetName, val, 0, count);
}
```

(b) removed field `offset`

Listing 4.3: Method `String.getBytes()`

Internally, the VM uses C++ methods for allocating and manipulating string objects, which also reference the field `offset`. These methods are for example invoked during class loading, when string objects are allocated for the string constants in the constant pool. Furthermore, `offset` is also used in several intrinsic methods of the JIT compiler. To achieve a complete removal of the field `offset`, these parts of the VM must be modified. The three described cases for the changes on the Java source code also apply to the methods within the VM. Listing 4.4 shows a C++ method that calculates the length of an UTF-8 encoded string.

```
int java_lang_String::utf8_length(oop java_string) {
  typeArrayOop value  = java_lang_String::value(java_string);
  int          offset = java_lang_String::offset(java_string);
  int          length = java_lang_String::length(java_string);
  jchar* position = (length == 0) ? NULL : value->char_at_addr(offset);
  return UNICODE::utf8_length(position, length);
}
```

(a) original

```
int java_lang_String::utf8_length(oop java_string) {
  typeArrayOop value  = java_lang_String::value(java_string);
  int          length = java_lang_String::length(java_string);
  jchar* position = (length == 0) ? NULL :
                    java_lang_String::char_base(java_string);
  return UNICODE::utf8_length(position, length);
}
```

(b) removed field `offset`

Listing 4.4: Java HotSpot^{TM} VM method `utf8_length`

4.2 Removing the Field `hashcode`

A string's hash code is computed from its characters using the hash code algorithm that is specified in the documentation of the class `String`. Because the hash code depends on the string characters, two strings with the same characters have the same hash code. When the hash code of a string object is computed for the first time, it is cached in the field `hashcode` to avoid recomputations. This is beneficial because the method `String.hashCode()` is invoked often if certain data structures like hashtables are used. However, the field `hashcode` uses another four bytes per string object, so that its removal would reduce the memory usage. Therefore, an alternative memory location must be found where the hash code can be cached.

As described in Chapter 2.6 on page 14, each Java object has a header of two machine words. The first machine word is the *mark word* and caches the identity hash code. The identity hash code is a random number that is not guaranteed to be unique for each Java object. Therefore, it should not cause any problems if the identity hash code is equal to the string hash code for optimized string objects. This allows to use the same memory location for caching the string hash code and the identity hash code. Therefore, the field `hashcode` can be removed without loosing the advantages of caching the hash code.

On 32-bit architectures, the identity hash code is truncated to 25 bit when it is stored in the *mark word*. Now, this also applies to the string hash code. The evaluation shows

that the truncation does not have a negative impact on the performance, but it violates the specification of the hash code algorithm of the class `String`. Therefore, the string optimization is evaluated with and without the removal of the field `hashcode`. For this, a command-line option is added to the Java HotSpot^{TM} VM to activate this additional optimization on demand. On 64-bit architectures the optimization complies with the specification of the hash code algorithm because the object header is large enough to hold the hash code without truncation. Two different approaches for removing the field `hashcode` are implemented and evaluated:

- If the removal of the field `hashcode` is activated, the method `String.hashCode()` is removed from the method table of the class `String`. The removal causes the base class' `hashCode()` method to be invoked. The base class for strings is `Object`, which has the native method `Object.hashCode()`. Alike the method `System.identityHashCode()`, this method calls into the VM to execute the identity hash code algorithm there.

  This algorithm is modified so that the string hash code is computed for optimized strings instead. After the computation, the object header's *mark word* is merged with the hash code. If the string is currently locked, this operation is expensive because an inflated monitor is required to store the hashcode. In the evaluation (see Chapter 5), it turned out that this implementation has a negative impact on the performance of some benchmarks.

- The Java method `String.hashCode()` is changed for this implementation, as shown in Listing 4.5: the method computes the hash code, truncates it to 25 bit and does not do any caching as long as the method is executed by the interpreter. Therefore, the hash code must be computed upon each invocation. This could only be avoided by introducing a new bytecode for accessing the *mark word*. If the method is compiled or inlined, an intrinsic method is used instead. It calculates the hash code once and caches the result in the *mark word*. As long as a string object is locked, the hash code caching is disabled and the hash code is always computed because this is less expensive.

  It is still necessary to modify the identity hash code algorithm within the VM because the method `System.identityHashCode()` could be invoked on a string object before the string hash code is cached. Without the modification, the method `System.identityHashCode()` would store a random number as the identity hash code. The later invoked method `String.hashCode()` would find the cached identity hash code and would return it as the string hash code.

```java
public int hashCode() {
  int h = hashcode;
  if(h == 0) {
    int off = offset;
    char val[] = value;
    int len = count;

    for(int i = 0; i < len; i++) {
      h = 31*h + val[off++];
    }
    hashcode = h;
  }
  return h;
}
```

(a) original

```java
public int hashCode() {
  int h = 0;
  int len = count;

  for(int i = 0; i < len; i++) {
    h = 31*h + value[i];
  }
  h = h & 0x1FFFFFF;
  return h == 0 ? 0xBAD : h;
}
```

(b) optimized hash field

Listing 4.5: Method `String.hashCode()`

The removal of the field `hashcode` also has a disadvantage: if the hash code is computed for a string object, this object can no longer use biased locking (see Chapter 2.6 on page 14). However, strings are not commonly used for synchronization.

4.3 Character Access

To access the characters of an optimized string, new bytecodes are necessary because the character array is merged with the string object. The string characters are declared as private and cannot be accessed directly from outside the class `String`, so that a method must be used instead. Therefore, the new bytecodes for accessing the characters of optimized strings are only needed within the class `String`. This has the advantage that no existing program must be modified or recompiled for accessing optimized strings. The two new bytecodes for loading and storing a string character are shown in Figure 4.1.

- `scload`: The bytecode loads a character from an optimized string, and is similar to the bytecode `caload` that is used to access 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 character in an optimized string, and is similar to the bytecode `castore` that is used to store a character in a character array. In comparison to the bytecode `scload`, the character to be stored is expected as an additional operand on the stack.

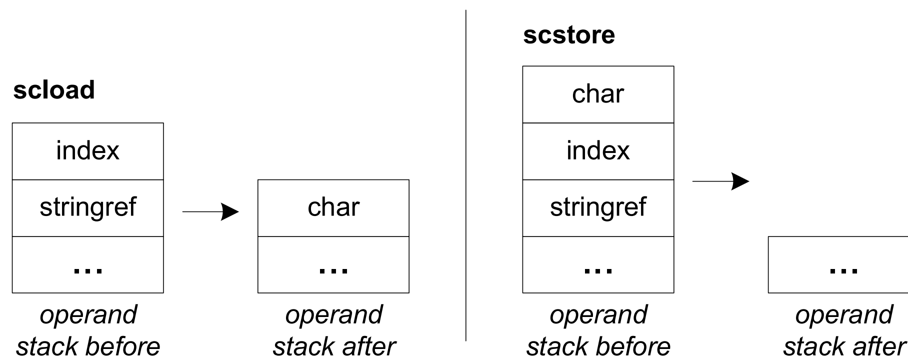
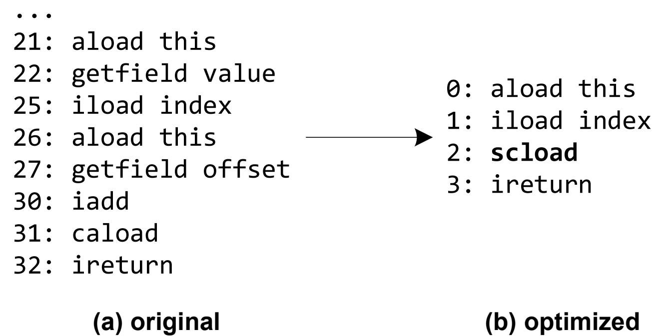


Figure 4.1: New bytecodes for accessing a character of an optimized string

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

- Figure 3.1 on page 22 and Figure 3.2 on page 23 show that the displacement for accessing characters of arrays and optimized strings is different. To access the first character of a character array, the correct displacement is 12 bytes. For an optimized string object, it is 12 or 16 bytes, depending on the optimization level.
- When a character array is accessed, a bounds check is performed. If it fails, an `ArrayIndexOutOfBoundsException` is thrown. Optimized strings also need bounds checks, but it is a convention for the class `String` to throw a `StringIndexOutOfBoundsException` if the check fails.

Original strings perform two bounds checks when they are accessed: an explicit bounds check is implemented in the class `String` that throws a `StringIndexOutOfBoundsException` if it fails. A second implicit bounds check is performed when the character array is accessed. Because the new bytecodes throw a `StringIndexOutOfBoundsException` if the bounds check fails, the explicit bounds checks can be removed from all string methods. This reduces the number of bytecodes necessary for accessing a string character and speeds up the execution. Furthermore, the new bytecodes address the characters directly via the string object, so that no character array pointer must be loaded and dereferenced. This further reduces the number of necessary bytecodes. The total size of the class `String` is reduced by approximately 6% because of these advantages. Listing 4.6 compares the bytecodes of the original method `String.charAt()` with the bytecodes of the optimized method. The first 20 bytes of the original version perform the explicit bounds check and are omitted for the listing.



Listing 4.6: Bytecodes of the method `String.charAt()`

4.4 String Allocation

The allocation of optimized string objects is more difficult than accessing their characters, because the class `String` has 16 constructors that must be preserved. Otherwise, existing programs would break. The allocation and the initialization of an object are performed by two different bytecodes in Java. For the allocation, the size of the object must be known. Because of merging the character array with the string object, it would be necessary to know the number of characters that will be stored in the optimized string. However, this number of characters is usually computed in the string constructor and therefore not available at the time of allocation. For solving this problem, three approaches were evaluated:

- The number of characters could be calculated by a string length computation method that is executed before the allocation. However, this method executes nearly the same code as the constructor and introduces a significant overhead. Furthermore, it needs the same parameters as the constructor and would therefore remove them from the stack, although they are still needed by the constructor. Therefore, the parameters would have to be pushed on the stack a second time, or a special return bytecode that leaves the old parameters on the stack would have to be implemented.
- The allocation could push a dummy string reference on the stack that is patched to the real string object when the number of characters is known. Stack-management instructions [21] such as `dup` could duplicate the dummy reference several times. Therefore, it would be necessary to keep track of the dummy references in order to patch each occurrence.
- The finally implemented approach replaces the string constructors with factory methods. For each constructor, a corresponding static factory method is introduced that takes the same arguments as the constructor. Such a method calculates the string

length, allocates the string object, and initializes it. Invoking the factory method therefore replaces the string allocation and the constructor invocation. Because this is all done in one method, it does not add any overhead. Listing 4.7 shows the factory method that replaces the constructor `String(char[])`. However, this approach has the disadvantage that all string allocations must be modified. The details of this modification are presented in the next section.

```
private static String allocate(char value[]) {
   int len = value.length;
   String string = newstring(len);
   stringcopy(value, 0, string, 0, len);
   return string;
}
```

Listing 4.7: A factory method

Figure 2.4 on page 12 shows the currently used bytecodes for instance and array allocation. An optimized string object has fields like an instance and a variable size like an array. Therefore, no existing bytecode is suitable for the allocation of an optimized string object. Instead, the bytecode `newstring` is introduced, which is similar the bytecode `newarray`. As shown in Figure 4.2, the bytecode `newstring` expects the string length on the operand stack. A second operand, such as the element type for arrays, is not necessary because the number of fields for an optimized string object is fixed and only characters can be stored in the variable part. This knowledge is used to compute the total size of the optimized string object, so that the allocation can be performed. The bytecode `newstring` is exclusively used within the class `String` and implements the italic part in Listing 4.7.

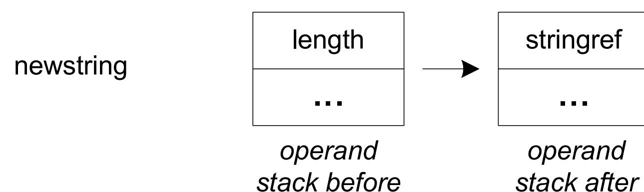


Figure 4.2: Stack before and after the execution of the bytecode `newstring`

4.5 Bytecode Rewriting

The allocation and initialization of optimized strings is performed by factory methods. However, all classes that allocate string objects still try to invoke the no longer existing string constructors and therefore must be rewritten. Because the source code is usually not available, this must be done at run time. The rewriting adds a negligible overhead because it is done only once for each class after class loading.

Figure 4.3 shows the separate rewriter component that inspects each class after class loading. All methods that allocate string objects are rewritten. The current implementation of the bytecode rewriting is a heuristic approach and performs the following three steps, also shown in Listing 4.8.

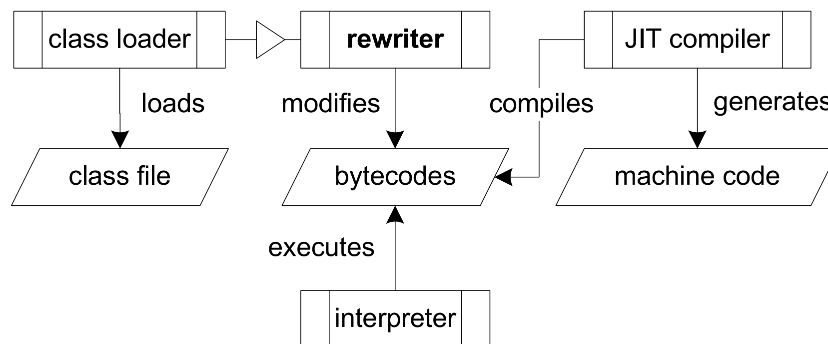


Figure 4.3: Bytecode rewriter component

1. Because the factory methods are used for the allocation and initialization of optimized string objects, the original string allocation is removed by replacing it with no operation (`nop`) bytecodes. The bytecode `new` and its arguments have a total size of three bytes, and therefore three `nop` bytecodes are used as a replacement. A complete removal of the affected instructions is not recommended because it would cause side effects on all jumps within the method. The `nop` bytecodes are ignored anyway when the method is compiled. The bytecode `new` would place a reference to an uninitialized string object on top of the operand stack. This does not happen anymore because of the replacement.
2. Because arbitrary code can be between the string allocation and string initialization, bytecodes like stack-management instructions might use the reference to the uninitialized string object. However, this reference does not exist anymore because the string allocation has been replaced with `nop` bytecodes. Therefore, all bytecodes that try to operate on the uninitialized string object must be replaced, removed or

```
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;
    }
  }
}
```

Listing 4.8: Bytecode rewriting heuristic

rearranged. For this, it would be necessary to parse the bytecodes and to build a representation where the affected instructions can be identified and modified.

The current implementation is prototypical and handles only the two most common stack management instructions `dup` and `pop`. Because the other stack management instructions are rarely used, the implementation is still complete enough for running nearly all Java programs including the various benchmarks presented in Chapter 5. However, bytecode-optimized applications might use other bytecodes that are not handled by the current implementation. Although the current implementation does its best about error detection, it cannot guarantee that all not yet supported instructions are reported as an error.

3. The last step of the rewriting process replaces the constructor invocation with the invocation of the static factory method. This rewriting step could be implemented in two different ways:

   - The factory method could be added to the constant pool and the index of the invoked method could be adjusted. However, this does not work for native code that uses the JNI because this code is not available as Java bytecode.
   - If the invoked method's constant pool index still points to the string constructor, it is possible to modify the method resolution within the Java HotSpot™ VM. If a string constructor is to be resolved, the factory method is returned instead. Applications that use the JNI must retrieve a method identifier before

invoking a method. Retrieving the method identifier internally uses the method resolution. Therefore, this approach also works for the JNI and because of this advantage, this approach is implemented.

Additionally, the method invocation bytecode must be changed from `invokespecial` to `invokestatic`. No other changes are necessary because the factory methods have the same arguments as the constructors.

Listing 4.9 (a) shows a simple Java method. The method allocates and returns a new string that is initialized with a character array. The bytecodes of the unoptimized version are shown in Listing 4.9 (b): the bytecode `new` allocates an original string object, for which the size is statically known. This allocation pushes a reference to the string on the stack, which is duplicated by the stack management instruction `dup` because it is needed both for the constructor invocation and the method's return. Then, a reference to the character array `ch` is pushed on the operand stack as the constructor's parameter. The constructor is invoked to perform the initialization and the initialized string object is returned.

```
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 nop nop
13: nop
14: aload ch
15: invokestatic constructor
18: areturn
```

(c) optimized bytecode

Listing 4.9: Example for bytecode rewriting

For allocating the optimized string object, the rewriter must modify the original bytecodes. The string allocation and the subsequent reference duplication are replaced by `nop`

bytecodes, and the bytecode `invokespecial` is replaced by `invokestatic`. The result is shown in Listing 4.9 (c). Upon execution, the method resolution returns the factory method instead of the string constructor and therefore the static factory method is invoked by the bytecode `invokestatic`.

4.6 Java Programming Language Compiler

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 JVM's bootstrap classpath to override the default implementation of the class `String`. Without the modified version of javac, the original class `String` would have to be transformed to the optimized one at runtime. This is less flexible and more effort to implement. During the code generation of javac, all necessary data structures exist so that the old bytecodes can be replaced by the newly introduced ones. This version of javac is only used to create the optimized string class file and is not used for compiling any other Java source code.

Because the transformation is not performed at run time, it is possible to provide Java-like source code for the optimized class `String`. This has the advantage that the optimized class is easier to understand and to change. However, the semantics of some statements are adjusted to express the newly introduced bytecodes. The string's field `value`, 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. Therefore, the reader must be conscious that although this field looks and is used like a character array, it is something that cannot be fully expressed in the Java language. The field `value` also exists in the generated optimized class file, but it is filtered when the class file is loaded by the VM.

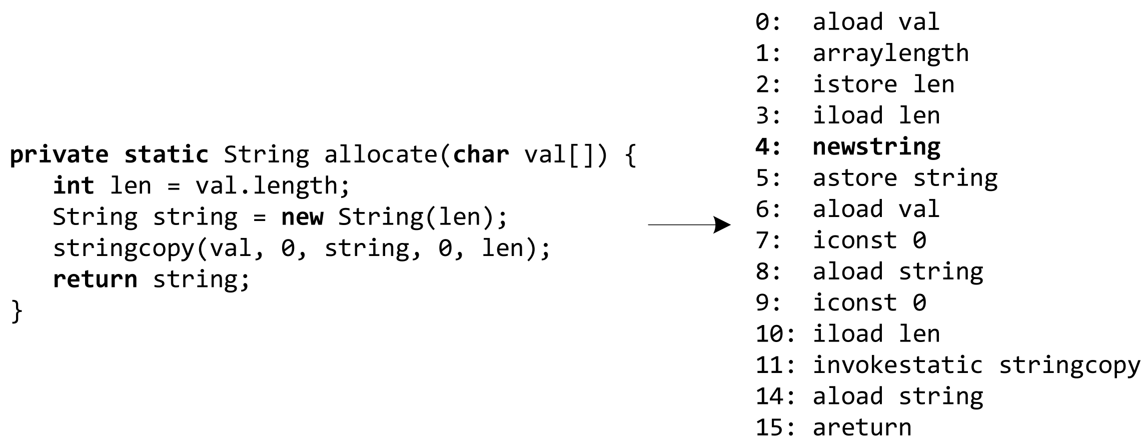
All existing constructors are replaced by the factory methods described in Chapter 4.4 and one synthetic constructor is added. This synthetic constructor has the string length as its only parameter and is used to model the bytecode `newstring` in the Java source code. The modified version of javac uses 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 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 argument. 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 Listing 4.6 on page 35. The bytecodes for a compilation of a factory method are shown in Listing 4.10.



```
private static String allocate(char val[]) {
   int len = val.length;
   String string = new String(len);
   stringcopy(val, 0, string, 0, len);
   return string;
}
```

```
0:  aload val
1:  arraylength
2:  istore len
3:  iload len
4:  newstring
5:  astore string
6:  aload val
7:  iconst 0
8:  aload string
9:  iconst 0
10: iload len
11: invokestatic stringcopy
14: aload string
15: areturn
```

Listing 4.10: Factory method compiled with the modified version of javac

4.7 Heap Management Class Hierarchy

The Java HotSpot^{TM} VM uses so called *oops* for the heap management. Each object on the heap is an instance of one of the *oop* classes. Figure 4.4 shows the class hierarchy of the *oops* relevant for this master's thesis. The abstract class `oop` serves as the base class for all objects on the heap. Each object on the heap has an object header that consist of the *mark word* and the class descriptor. The *mark word* is a bitfield and the class descriptor is modeled by the class `klassOop`. The `klassOop` is a container for one of the subclasses of `Klass`, which are the actual class descriptors for specific data types, e.g. `arrayKlass` for Java arrays. The class descriptor is used for the object size calculation, which is essential for the garbage collector to iterate over all objects on the heap. Furthermore, the class descriptor object holds additional metadata and is used for reflection.

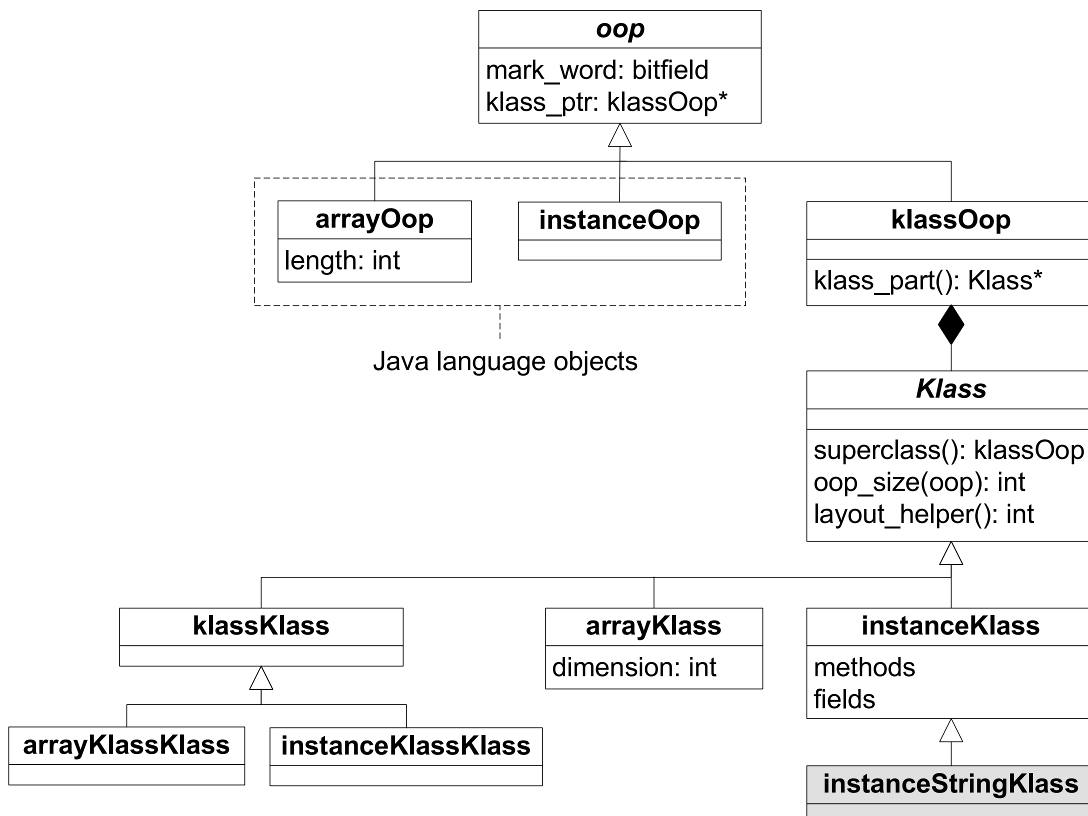


Figure 4.4: Simplified overview of the used *oops* in the HotSpot$^{\text{TM}}$ VM

The specific *oop* data structure necessary for the allocation and the management of a `StringBuffer` object is illustrated in Figure 4.5. For the allocation of `StringBuffer` object, the class `StringBuffer` must be loaded. This allocates a `klassOop` that serves as a container for an `instanceKlass` object. The `instanceKlass` object is the class descriptor for all `StringBuffer` objects and stores the information about the fields and methods of this class. For different Java classes, the `instanceKlass` objects differ in their size because Java classes have different numbers of fields and methods. Therefore, `instanceKlass` objects need their own class descriptor, for which `instanceKlassKlass` is used. Because `instanceKlassKlass` has a fixed size, it uses `klassKlass` as its class descriptor. `klassKlass` has the same fixed size and can therefore use itself as its own class descriptor. After the class loader finished loading the class `StringBuffer`, the actual `StringBuffer` object can be allocated. Because `StringBuffer` is a Java class, an `instanceOop` object is allocated that references the `StringBuffer` class descriptor. For arrays, the used classes are similar: an array on the heap is represented by an `arrayOop` object. This object has the array length as an additional field and references an `arrayKlass` object as its class descriptor. A Java programmer directly allocates only `instanceOop` and `arrayOop` objects.

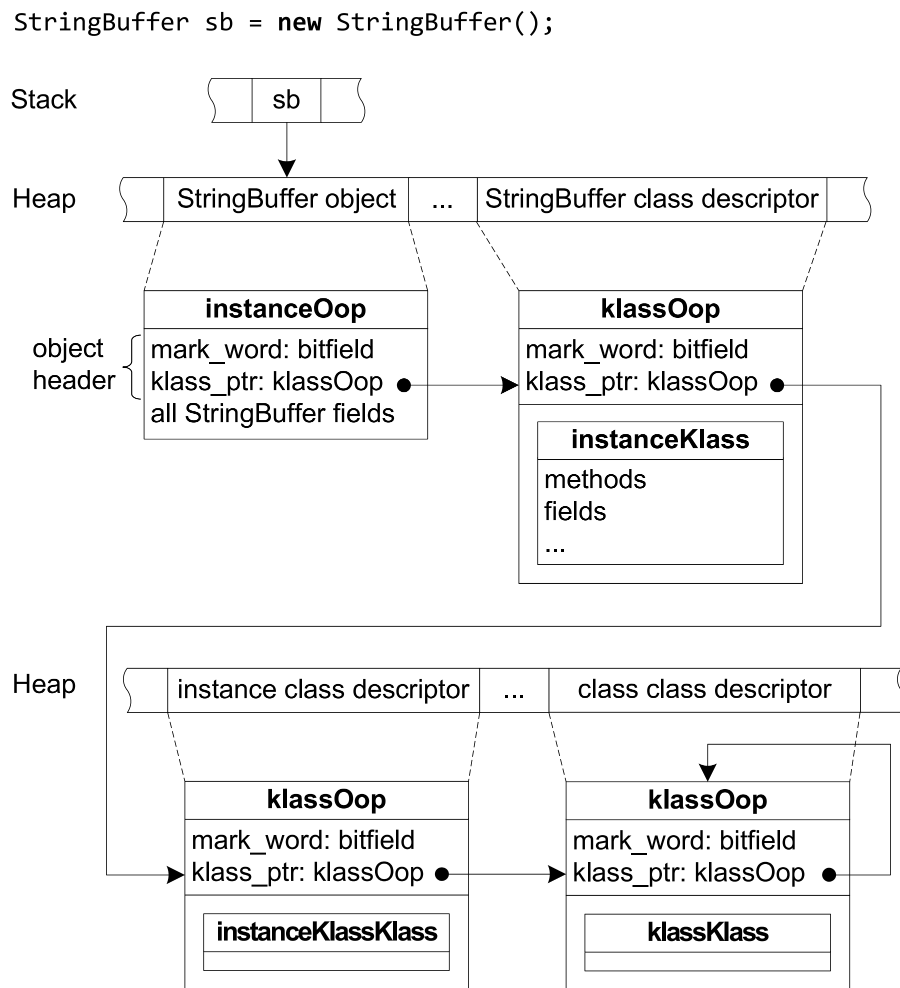


Figure 4.5: Example of all required *oops* for a `StringBuffer` object

For optimized strings, the class `instanceStringKlass` is introduced. This class inherits from `instanceKlass` because strings have all characteristics of instances (i.e. fields and methods). However, the separate class is still necessary because the size computation and the object allocation are different for optimized strings.

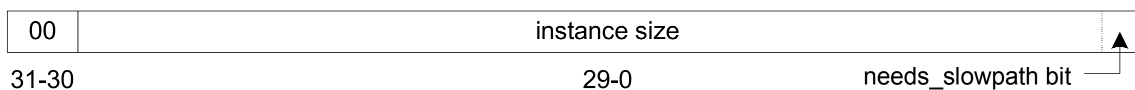
4.8 Layout Helper

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 calculated. 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™ 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 instances and arrays. However, these two are the majority on the heap because no other objects can be allocated directly from Java source code. The layout helper stores various information based on its bit pattern, as shown in Figure 4.6. The highest significant bit determines if an object is an instance or an array. Depending on this, the remaining bits are interpreted differently. The layout helper of instances classes contains the statically known size. For arrays, more information is necessary for computing the size: one bit is used to distinguish object arrays and type arrays, the others are used to store additional data such as the header size, the array element type and the element size.

(a) layout helper of instance classes



(b) layout helper of array classes

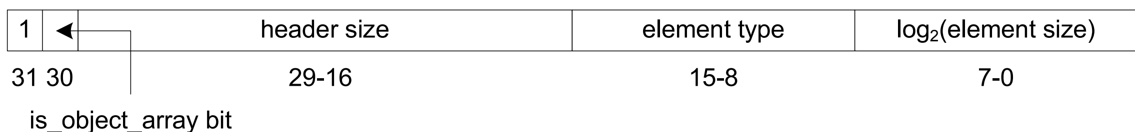


Figure 4.6: Bit patterns of the original layout helper

For all objects that are neither instances nor arrays, the layout helper is zero. This indicates that the virtual method must be used for the size computation. The encoding of the layout helper allows efficient processing. To check if an object is an instance, the instruction `test` can be used on IA-32 architectures [15]. This instruction does not need a result register because it only sets the processor flags that are queried for conditional jumps.

Listing 4.11 shows the code pattern for the fast object size calculation that uses the layout helper. At the beginning, the layout helper is once queried from the class descriptor. It is always assumed that the current object is an instance because this is true for the majority of the objects on the heap. The layout helper's encoding allows to obtain the instance

```
int size(Klass* klass) {
  int lh = klass->layout_helper();
  // assume that lh is the instance size
  int s  = lh >> LogHeapWordSize;

  if(!is_instance(lh)) {
    if(is_array(lh)) {
      s = compute_array_size(lh);
    } else {
      s = klass.size(); // call virtual method
    }
  }

  return s;
}
```

Listing 4.11: The original fast object size computation

size in heap words by just shifting the layout helper. The next step checks if the current object really is an instance. If so, the already computed size can be returned immediately, otherwise it is checked if the object is an array. If this is the case, the array element type, the array length and the header size are used to compute the total array size. If the object is neither an instance nor an array, the virtual method is invoked for the size computation.

The string optimization causes some problems with this existing infrastructure because optimized strings are instances that do not have a statically known size. Therefore, several modifications are necessary to support the fast size computation for optimized string objects. Differentiating between instances, arrays and other objects is not sufficient. The new layout helper, shown in Figure 4.7, distinguishes between objects with a fixed size, objects with a computed size, and other objects. The header size and the element size are still byte aligned because this allows direct access on IA-32 architectures. Separate bits are used to store if an object is an instance or an array. This has the advantage that the layout helper could also be used for VM internal objects that are neither instances nor arrays. For this purpose, the length offset is introduced for the computed-size layout helper. It could be used to determine the location, where the object's number of elements is stored. However, the current implementation does not use the layout helper for objects other than instances, arrays and strings. Furthermore, the length offset is also currently not used because the field `length` of arrays has exactly the same offset as the field `count` for optimized strings. Therefore, the length offset is grayed out in Figure 4.7.

The changes to 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 instances and arrays. However, as shown in Listing 4.12, only few changes are necessary for the code pattern that is used for the fast object size calculation.

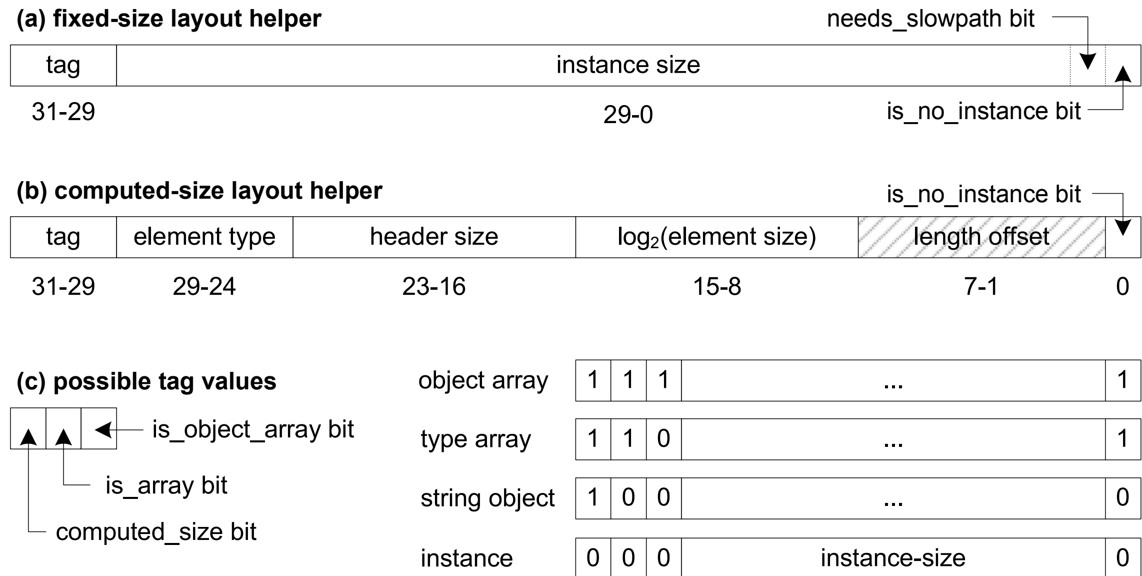


Figure 4.7: Bit patterns of the new layout helper

```
int size(Klass* klass) {
  int lh = klass->layout_helper();
  // assume that lh holds the object size
  int s  = lh >> LogHeapWordSize;

  if(!has_fixed_size(lh)) {
    if(size_must_be_computed(lh)) {
      s = compute_size(lh);
    } else {
      s = klass.size(); // call virtual method
    }
  }

  return s;
}
```

Listing 4.12: The new fast object size computation

4.9 Interpreter

The interpreter uses machine dependent code templates for the bytecode execution. These templates are specified in assembler language and are translated to machine code without applying any optimizations at the startup of the Java HotSpot^{TM} VM. To support the three new bytecodes, the interpreter requires appropriate code templates. Because the templates are machine dependent, they are currently only implemented for the IA-32 architecture.

Listing 4.13 shows the interpreter code patterns for the bytecodes `scload` and `scstore` that are used for loading and storing string characters. The implementation is similar to the bytecodes `caload` and `castore` that are used for accessing a character array. Some of the necessary operands are already received in registers because they are results from previous computations. Others must be popped from the operand stack. To ensure that the index, which is used for accessing the string object, addresses a character within the string, a bounds check is performed. If the check fails, the index does not address a valid character and an `StringIndexOutOfBoundsException` is thrown. In the last step, the string character is read or written and if necessary, the result is returned in a specific register.

```
void TemplateTable::scload() {
  // rdx: string, rax: index
  string_index_check(rdx, rax);
  load_unsigned_word(rbx, Address(rdx, rax, Address::times_2,
        java_lang_String::value_offset_in_bytes()));
  movl(rax, rbx);
}

void TemplateTable::scstore() {
  // rax: value, rdx: string
  pop_i(rbx); // get index
  string_index_check(rdx, rbx);
  store_unsigned_word(Address(rdx, rbx, Address::times_2,
        java_lang_String::value_offset_in_bytes()), rax);
}
```

Listing 4.13: Interpreter code patterns for `caload` and `castore`

For the bytecode `newstring`, an interpreter template is difficult to implement because the whole string allocation would have to be done in assembler. This would result in a code pattern similar to the one used for the bytecode `new`. However, the major problem is that the newly allocated string object must reference a valid class descriptor. When

the interpreter templates are translated to machine code, the class `String` is not yet loaded and therefore the string class descriptor does not exist and cannot be accessed. It would be possible to solve this problem by the early creation of a fixed string class descriptor. However, this would reduce the flexibility. The bytecode `newarray` has a similar problem and solves it by moving the allocation into a method of the interpreter runtime. The interpreter code template only passes the appropriate arguments to this method and performs the method invocation. This interpreter runtime method is not called during the translation of the interpreter templates, so that the access to the string descriptor does not cause any problems. Therefore, a similar approach is used to implement the allocation of optimized string objects, as shown in Listing 4.14.

```
void TemplateTable::newstring() {
  call_VM(rax, CAST_FROM_FN_PTR(address, InterpreterRuntime::newstring), rax);
}

void InterpreterRuntime::newstring(JavaThread* thread, jint size) {
  Handle obj = java_lang_String::create(size, CHECK);
  thread->set_vm_result(obj());
}
```

Listing 4.14: Interpreter code pattern for `newstring`

4.10 Just-in-Time Compiler

To achieve comparable evaluation results, the new bytecodes are also implemented for the IA-32 architecture of the client compiler. Multiple modifications and extensions are necessary to support optimized string objects. The changes to the layout helper also require some other changes in the client compiler. The client compiler uses a high-level and a low-level intermediate representation, as described in Chapter 2.4.2 on page 10. Both representations are modeled as C++ class hierarchies. A small subset of the high-level intermediate representation (HIR) class hierarchy is presented in Figure 4.8.

The class `Instruction` is the abstract base class for all HIR instructions. The whole class hierarchy consists of 15 abstract classes and 47 classes that represent specific HIR instructions. For the allocation of optimized string objects, the class `NewString` is added to the class hierarchy. Because accessing a string character is similar to accessing a character array, the functionality of the classes `StoreIndexed` and `LoadIndexed` is extended to support optimized string objects. For this purpose, a boolean flag is added to the abstract class `AccessIndexed` that is the base class of `StoreIndexed` and `LoadIndexed`. The

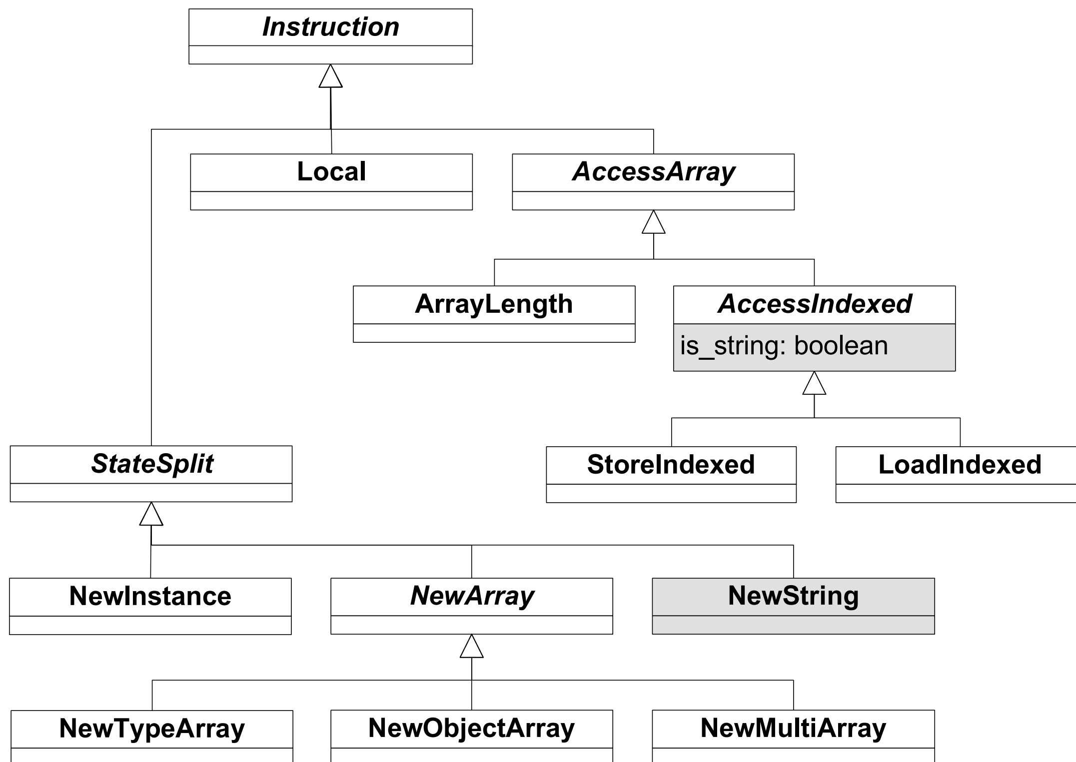


Figure 4.8: Subset of the HIR class hierarchy

subsequent low-level intermediate representation (LIR) generation uses this boolean flag to emit the correct LIR instructions. Because the same HIR instructions as for accessing an array are used for accessing optimized strings, all implemented optimizations for these instructions (e.g. bounds check elimination) are automatically applied to optimized strings as well.

Figure 4.9 shows a subset of the LIR class hierarchy that has 20 classes. The base class for all LIR instructions is `LIR_Op` that stores a LIR operation code. More than 80 LIR operation codes are used to model the actual instructions. For example, the instruction `push` uses a `LIR_Op1` object, where the LIR operation code is set to `push`. Therefore, a LIR class can be used for multiple instructions, which simplifies the class hierarchy. The LIR operation codes are used by the subsequent operations on the LIR, e.g. the code generation uses them to emit the correct machine code.

The class `LIR_OpAllocArray` is extended so that it can also be used for the allocation of optimized string objects. For this purpose, a new LIR operation code is introduced. For accessing the characters of optimized string objects, no additional LIR classes or LIR

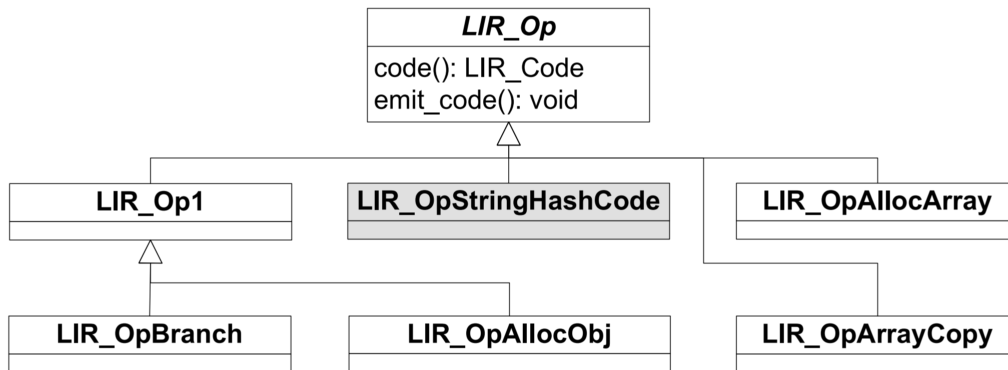


Figure 4.9: Subset of the LIR class hierarchy

operation codes are necessary because the HIR can be translated to already existing LIR instructions.

The method `System.arraycopy()` can no longer copy characters from or to an optimized string because the characters are not stored in a character array anymore. However, `System.arraycopy` is faster than a loop in Java because some bounds and type checks are omitted. Therefore, similar methods are introduced for optimized string objects: `String.stringToStringCopy()`, `String.stringToCharCopy()` and `String.charToStringCopy()`. These three methods copy the characters between two strings or between a character array and a string. A method `charToCharCopy()` is not necessary because this is implemented by `System.arraycopy()`. The new methods are compiler intrinsics that share most of its code with `System.arraycopy()`. Therefore, the class `LIR_OpArrayCopy` is also used to represent these new methods in the LIR. However, each method has its own LIR operation code that the code generation uses to emit the appropriate machine code.

For the removal of the field `hashcode`, a compiler intrinsic is used for the method `String.hashCode()`. To represent this intrinsic in the LIR, the class `LIR_OpStringHashCode` and a new LIR operation code are added. The actual intrinsic method is specified in assembler and performs the steps shown in Listing 4.15. If the string is unlocked, its *mark word* might already store a hash code. If no valid hash code is present, or the string is not in the unlocked state, the hash code must be computed. After the computation, the hash code can be only stored in the *mark word* if the string is either unlocked or biasable.

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

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

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

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

  return hashcode;
}
```

Listing 4.15: `String.hashCode()` compiler intrinsic

4.11 Elimination of Explicit String Copying

Copying strings by invoking the constructor `String(String)` is nearly useless because strings are immutable. For original strings, the internal character array is trimmed on an explicit copy, i.e. if only a part of the referenced character array is in use, a new character array is allocated where the used characters are copied to. Optimized strings always use all their characters, so that explicit copies are useless. However, these copies still cannot be removed easily because of two reasons:

- **Object equality:** If no explicit string copy is allocated, the semantics of string object equality checks can change. This might influence the program behavior in an unexpected way.
- **Synchronization**: In Java, any object can be used as a monitor for synchronization. When a string object is used as a monitor, the program behavior might change if no explicit string copy is allocated.

For removing explicit string copying, these cases would have to be detected. However, both cases are uncommon and therefore the elimination of explicit string copying was implemented without a detection mechanism. Because this additional optimization is unsafe, it is evaluated separately. For the allocation of optimized string objects, static factory methods are used instead of constructors. Therefore, the implementation is easy: the fac-

tory method `String.allocate(String)` returns the string parameter without allocating a copy, as shown in Figure 4.10.

```
private static String allocate(String original) {
  int len = original.length();
  String string = new String(len);
  stringcopy(original, 0, string, 0, len);
  return string;
}
```

(a) explicit copying

```
private static String allocate(String original) {
  return original;
}
```

(b) elimination of explicit string copies

Figure 4.10: Explicit string copying factory method

Chapter 5

Evaluation

The optimizations are implemented for the early access version b24 of Sun Microsystems' JDK 7 [29]. Because several parts of the implementation are platform dependent, the string optimization is currently only implemented for the interpreter and the client compiler of the IA-32 architecture. The SPECjvm98 [26], SPECjbb2005 [27], and the DaCapo [3] benchmarks are used for the evaluation. Separate benchmark runs are executed for measuring the performance and the number of allocated bytes. Additionally, two different machines are used for the measurements:

- **Laptop:** An Acer Aspire 5652 Laptop with an Intel Core Duo processor with 2 cores running at 1.6 GHz, 2 MB L2 cache, 1 GB main memory, and with Windows XP Professional as the operating system, is used as a typical consumer machine. Because this system changes the processor's clock rate even while benchmarking, it is permanently throttled to its minimum: 1.0 GHz per core.

  For accurate results, each measurement is repeated five times for the benchmarks SPECjbb2005 and DaCapo, and 15 times for SPECjvm98. The mean values of these runs are reported as the results. Furthermore, each benchmark is executed with multiple heap sizes to evaluate the effect on the optimization.

- **Workstation:** 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, is used to evaluate the optimization on a more powerful machine. To achieve accurate results, each measurement was repeated 12 times and the mean values are reported.

The three benchmark suites were executed on both machines with multiple configurations:

- The unmodified Java 7 build b24 is the baseline configuration.
- The configuration “no offset” removes the field `offset` from the class `String`. The character array is still a separate object, so that character array sharing is possible for explicit string copies. The results indicate if the removal of the field `offset` is sufficient for a performance gain, or if the other optimizations have a larger positive impact. Although the removal of the field `offset` would save 4 bytes per string object, this advantage is voided because of the alignment of Java objects.
- The configuration “optimized” uses all presented optimizations except the removal of the field `hashcode`.
- The configuration “overhead” uses the same optimizations as the configuration “optimized”, but adds an artificial padding of 20 bytes per string object. This results in strings that have the same size as original strings including the character array. The results for this configuration show the impact of the optimizations without the advantage of reduced memory usage. Furthermore, it indicates how much character array sharing a benchmark uses: the difference between the number of allocated bytes of this configuration and the baseline are the bytes that are allocated additionally because character array sharing is precluded. In comparison with the configuration “optimized”, this configuration shows the impact of the reduced memory usage on the performance.
- The configuration “optimized hash field” uses all presented optimizations including the removal of the field `hashcode`. For removing the field `hashcode`, the implementation variant with the intrinsic method is used (see Chapter 4.2 on page 31). Because only the IA-32 architecture is evaluated, the hash code is always truncated to 25 bit. The SPECjbb2005 benchmark compares during its startup the hash code of a string object with a hardcoded value. If this value does not match, the benchmark stops the execution. Therefore, this check was removed. However, the removal does not influence the results because this check is only performed once during the startup of SPECjbb2005.
- The “slow hash field” configuration is reported additionally for the SPECjvm98 benchmark. It also uses all presented optimizations like the “optimized hash field” configuration, but implements the removal of the field `hashcode` differently. It removes the method `String.hashCode()` so that the base class method is invoked instead (see Chapter 4.2 on page 31). The configuration is only reported for the SPECjvm98 benchmark because the results does not show any significant difference results than the configuration “optimized hash field” for the other benchmarks.

The configurations "optimized", "overhead" and "optimized hash field" use the same modified VM. A specific configuration is selected by a command-line option. Because the configurations need different `String` class files, a jar file with the appropriate class `String` is prepended to the bootstrap classpath. The configurations baseline, "no offset" and "slow hash field" each use their own VM. The appropriate class `String` is directly contained in the file rt.jar.

5.1 SPECjbb2005

The SPECjbb2005 benchmark represents a client/server business application. Transactions are performed on a database that is held in the physical memory. A benchmark run is split up into measurements for a specific number of warehouses. Each warehouse is executed as a separate thread, i.e. if 8 warehouses are executed, 8 threads are performing transactions on the database. Furthermore, the size of the database increases with the number of warehouses so that less memory is available for executing the transactions. Therefore, the heap fills up faster and more garbage collections occur.

For a specific number of warehouses, transactions are executed for 240 seconds on the database. The total transaction throughput on the database is measured in SPECjbb2005 business operations per second (bops). The official SPECjbb2005 result is computed as the mean value of results for specific numbers of warehouses. Which numbers of warehouses are used for the computation depends on the number of processors/cores. The warehouses 2, 3, 4 and 5 contribute to the official result on the laptop, and the warehouses 4, 5, 6, 7 and 8 on the workstation. The benchmark uses lots of string operations, so the optimization can show its potential. Unless stated otherwise, a heap size of 1200 MB is used for all measurements on the workstation and 512 MB on the laptop. Because each number of warehouses is measured for 240 seconds, the number of allocated bytes depends on the performance. Therefore, a modified version that executes a fixed number of transactions is used for measuring the number of allocated bytes. All percentage values are relative to the baseline.

Figure 5.1 shows some SPECjbb2005 results for the workstation and the laptop. The performance shows a good increase for the optimized configurations because this benchmark is string intensive. The workstation shows a slightly larger performance increase than the laptop because of the improved garbage collection time. The serial garbage collection algorithm, which the Java HotSpot^{TM} VM uses per default, is a bottleneck if more cores are available. Each time a garbage collection is necessary, the whole program execution is stopped and the garbage collection is performed by only one processor/core. Because more

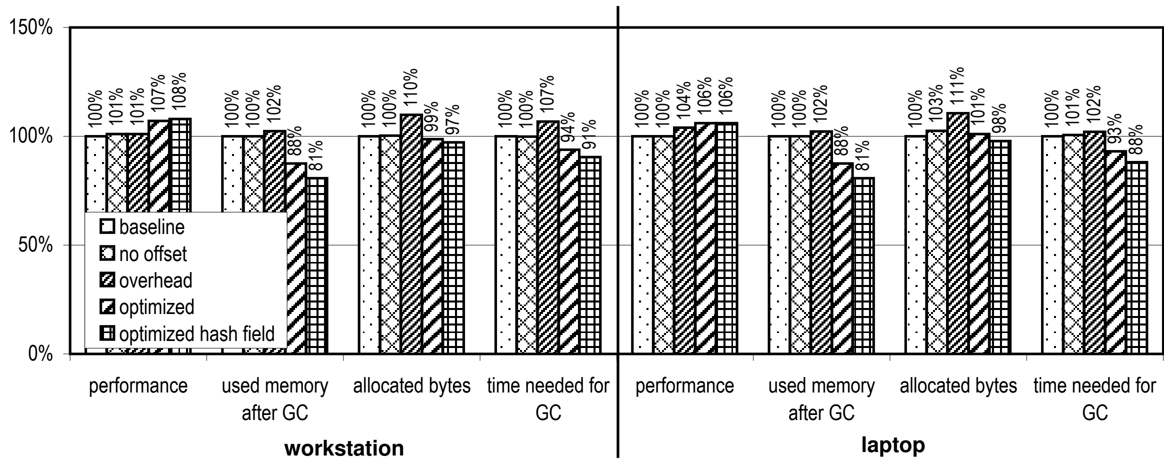


Figure 5.1: SPECjbb2005: benchmark results

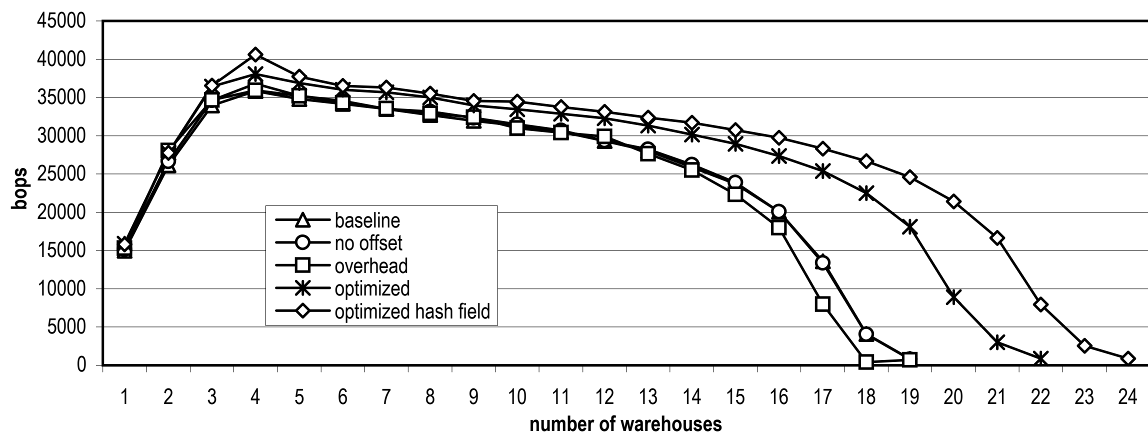


Figure 5.2: SPECjbb2005 workstation: performance (higher is better)

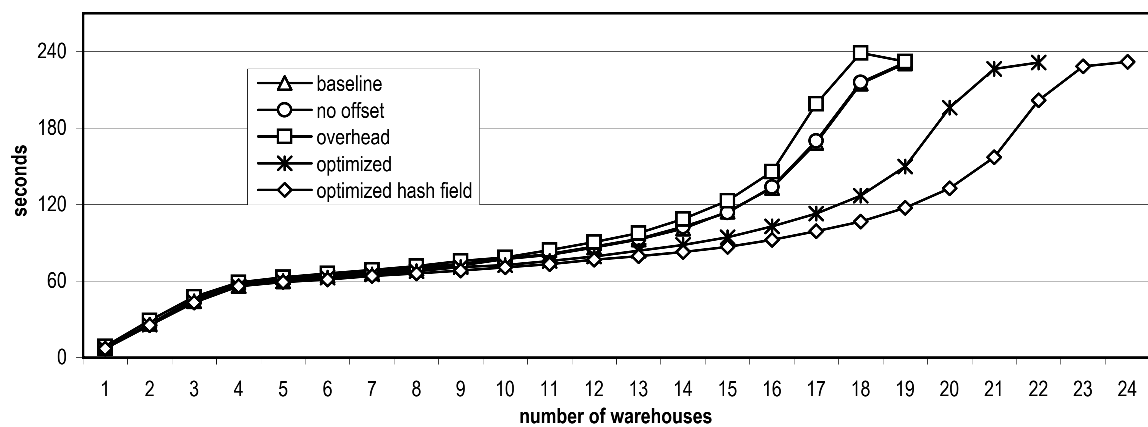


Figure 5.3: SPECjbb2005 workstation: garbage collection time (lower is better)

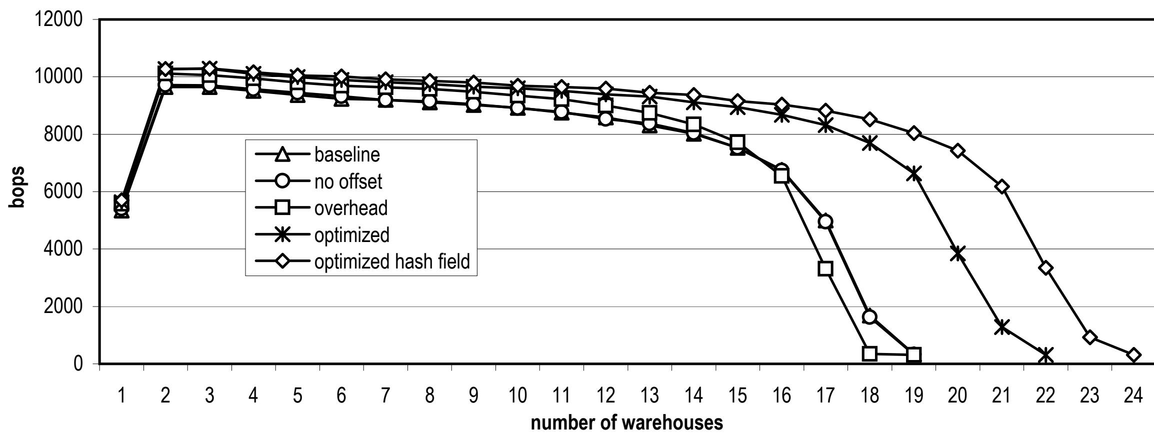


Figure 5.4: SPECjbb2005 laptop: performance (higher is better)

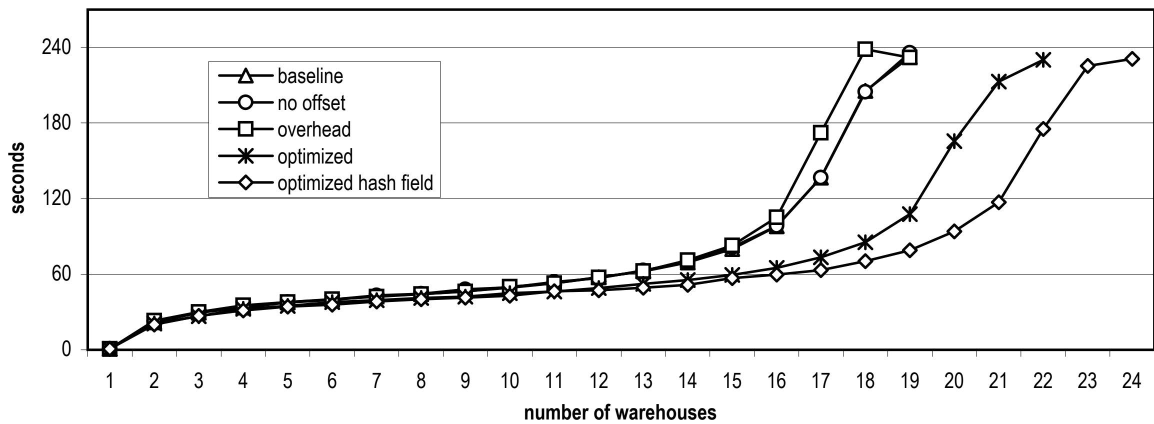


Figure 5.5: SPECjbb2005 laptop: garbage collection time (lower is better)

Figure 5.6: SPECjbb2005 laptop: average used memory after a full garbage collection (lower is better)

cores are unused during the garbage collection on the workstation, the reduced garbage collection time has a larger positive impact. This also indicates the configuration "overhead" that increases the garbage collection time. On the workstation, this results in a larger performance decrease than on the laptop. However, the performance is still higher than the baseline. Additional performance measurements with a heap size of 256 MB and 384 MB were performed on the laptop. These runs show similar results, but indicate that the performance increases for the optimized configurations in comparison with the baseline when the heap size is smaller. This is an effect of the reduced memory usage.

For measuring the number of allocated bytes independently of the performance, a modified version of SPECjbb2005 is used that executes a fixed number of transactions on four warehouses. The used memory after a garbage collection is an approximation of the minimum heap size of an application and is significantly improved for the optimized configurations. The total number of allocated bytes is only slightly reduced because the benchmark uses a high number of explicit string copies. The configuration "no offset" only uses character array sharing for explicit string copies and allocates approximately the same number of bytes as the baseline. The configuration "overhead" uses no character array sharing at all and allocates significantly more memory.

The time necessary for garbage collection is reduced for the optimized configurations because an optimized string needs only one object. Original strings consist of two distinct objects that both must be garbage collected. The slightly reduced number of allocated bytes also has a positive impact on the garbage collection time because the heap does not fill as fast and therefore fewer garbage collections are necessary.

Figure 5.2, 5.3, 5.4 and 5.5 show the performance and the garbage collection time for the laptop and the workstation. Each configuration was executed with a heap size of 512 MB, up to the number of warehouses where an `OutOfMemoryException` is thrown. The performance increases up to the peak number of warehouses, i.e. four warehouses for the workstation, two warehouses for the laptop. Subsequently, the performance decreases because the higher number of threads increases the overhead and because the memory resident database uses a larger part of the total available memory. Because the minimum heap size of the benchmark is reduced for the optimized configurations, it is possible to execute up to 24 instead of 19 warehouses. The performance is far better especially for a high number of warehouses. A clear correlation between the performance and the garbage collection time can be observed: the performance decreases as the garbage collection time increases. The workstation's garbage collection time increases much faster than the laptop's because the workstation executes far more transactions on the database, so that the heap fills faster and more garbage collections are necessary.

Figure 5.6 shows the average used memory after a full garbage collection on the laptop. No full garbage collection occurs during the run of the first warehouse. The memory usage increases with the number of warehouses, but the increase is significantly slower for the optimized configurations. The positive effect of removing the field `hashcode` can also be clearly seen for these results.

5.2 Elimination of Explicit String Copying for SPECjbb2005

About 19% of all string allocations in the SPECjbb2005 benchmark are explicit string copies. These string copies are only used for trimming the internal character array of original strings and can therefore be removed if the string optimization is active. The results in this section indicate which performance increase could be expected if the programmer knows that the allocation of explicit string copies is unnecessary.

Figure 5.7 shows some SPECjbb2005 results. Because of the elimination of explicit string copying, fewer objects must be allocated and fewer characters must be copied. In comparison to the results from the previous section, the performance increases significantly for the optimized configurations. However, the workstation shows a far larger performance increase than the laptop because of the improved garbage collection time. The used memory after a garbage collection and the number of allocated bytes are both significantly reduced for the optimized configurations. This also has a positive impact on the garbage collection time. The configuration “overhead” allocates approximately the same number of bytes as the baseline but the garbage collections are much faster because an optimized string needs only one object.

Figure 5.8, 5.9, 5.10 and 5.11 show the performance and the garbage collection time for the laptop and the workstation. As in the previous section, it is possible to execute up to 24 instead of 19 warehouses for the optimized configurations because of the benchmark’s improved minimum heap size. The reduced number of allocated bytes results in fewer garbage collections, which reduces the total garbage collection time. In comparison to the previous section, the performance of the optimized configurations is generally higher and decreases slower with an increasing number of warehouses.

Figure 5.12 shows the average used memory after a full garbage collection on the laptop. Again, no full garbage collection occurs during the run of the first warehouse. In comparison to the previous results, less memory is used after a full garbage collection. However,

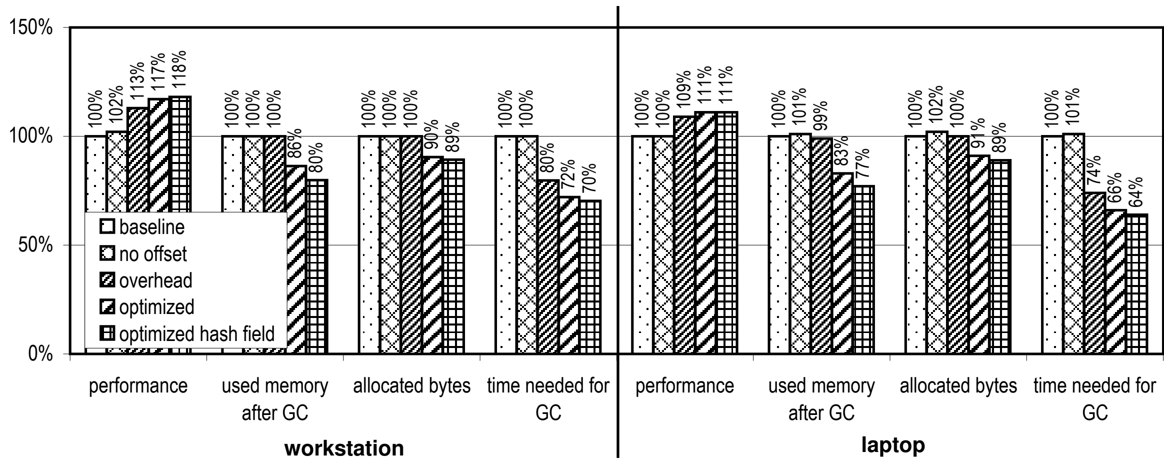


Figure 5.7: SPECjbb2005: benchmark results

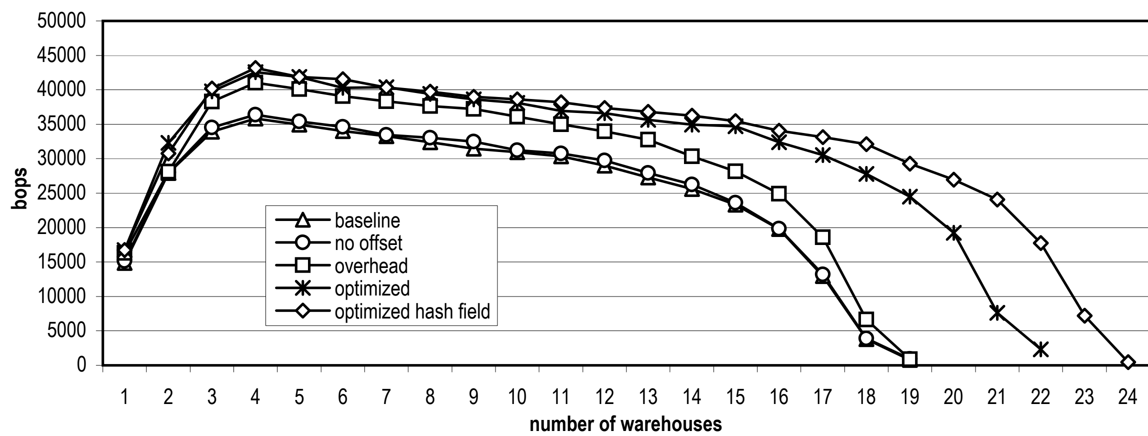


Figure 5.8: SPECjbb2005 workstation: performance (higher is better)

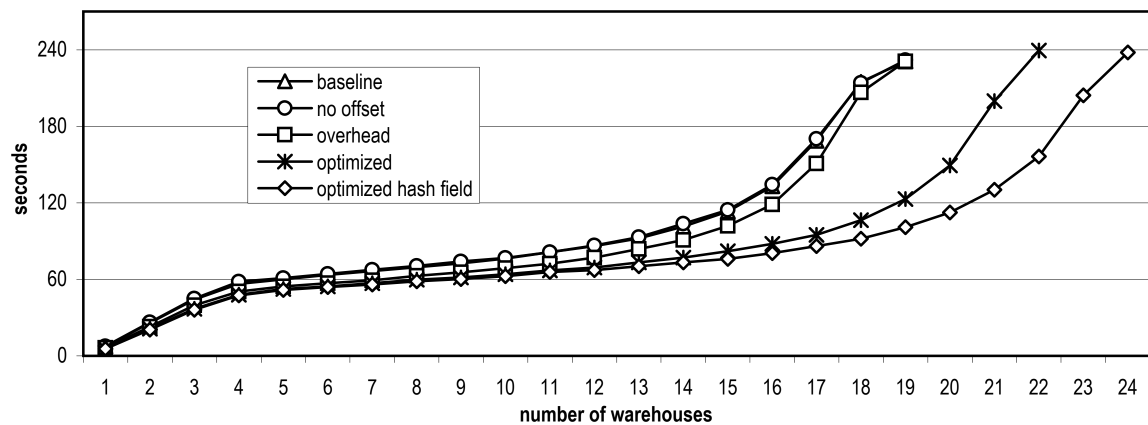


Figure 5.9: SPECjbb2005 workstation: garbage collection time (lower is better)

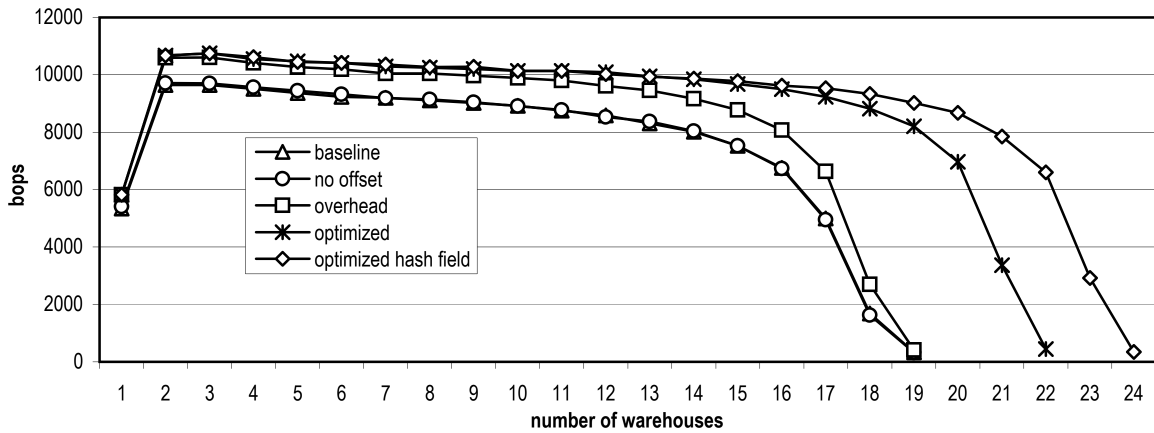


Figure 5.10: SPECjbb2005 laptop: performance (higher is better)

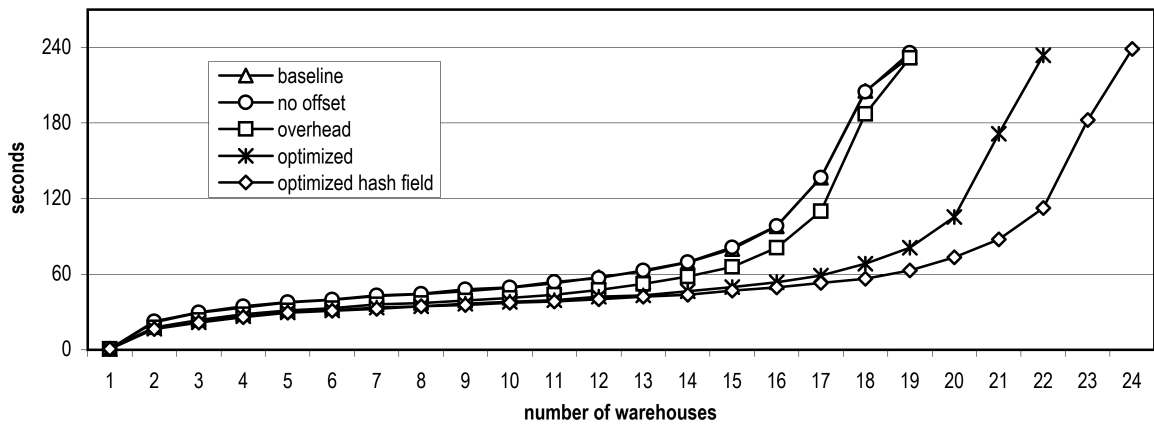


Figure 5.11: SPECjbb2005 laptop: garbage collection time (lower is better)

Figure 5.12: SPECjbb2005 laptop: average used memory after a full garbage collection (lower is better)

the reduction is not enough to execute more warehouses than without the elimination of explicit string copying.

5.3 DaCapo

Eleven object-oriented applications form the DaCapo benchmark suite release 2006-10-MR2. Each benchmark is executed five times in the same JVM, so that the execution time converges because all relevant methods are compiled. This results in a slowest run that measures the startup time of the JVM and a fastest run that indicates the achievable peak performance. These two results are presented for each benchmark. Additionally, the geometric mean of all benchmark results is presented. Unless stated otherwise, a heap size of 256 MB is used on both machines for all benchmarks.

The performance results for the DaCapo benchmark suite are presented in Figure 5.13 and Figure 5.14. The light bars refer to the slowest run, the dark bars to the fastest run. Both runs are shown on top of each other relative to the fastest run of the baseline. For nearly all benchmarks, the slowest and the fastest runs are improved on the workstation and the laptop. However, it depends on the benchmark if it shows a higher speedup on the laptop or on the workstation. The benchmark hsqldb shows a significant speedup for the workstation and a slight decrease on the laptop.

The benchmark chart shows on both machines the largest speedup. Some other benchmarks with a speedup on both machines are antlr, eclipse and jython. Because the configuration "overhead" shows a similar performance as the configurations "optimized" and "optimized hash field", the memory usage reduction has only a small impact on the results of this benchmark suite. The removal of the field `offset` alone is not sufficient for a significant speedup. It has a negative impact for some benchmarks, and a positive one for some others. Additional performance measurements with a heap size of 96 MB and 128 MB were performed on the laptop. These runs show no significant difference to the presented results.

The number of allocated bytes for the DaCapo benchmarks are shown in Figure 5.15. The benchmarks chart, luindex, and bloat need to allocate far less memory, but the benchmark jython shows a contrary result. Because character array sharing is no longer possible for optimized string objects, new strings are allocated and the characters must be copied. Nevertheless, the performance still shows a speedup. The configurations "no offset" and "overhead" need to allocate more memory than the baseline, which indicates that the DaCapo benchmark suite uses a significant amount of character array sharing. Furthermore,

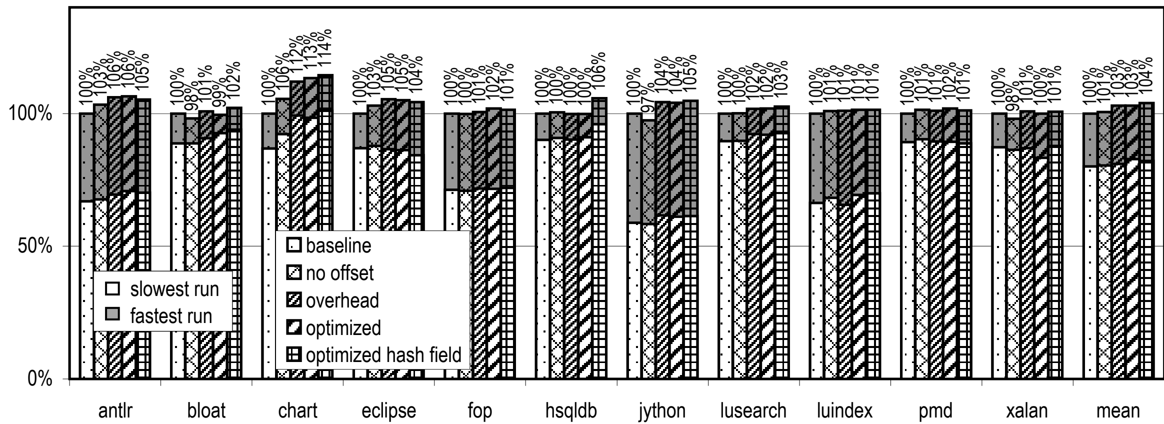


Figure 5.13: DaCapo workstation: performance (higher is better)

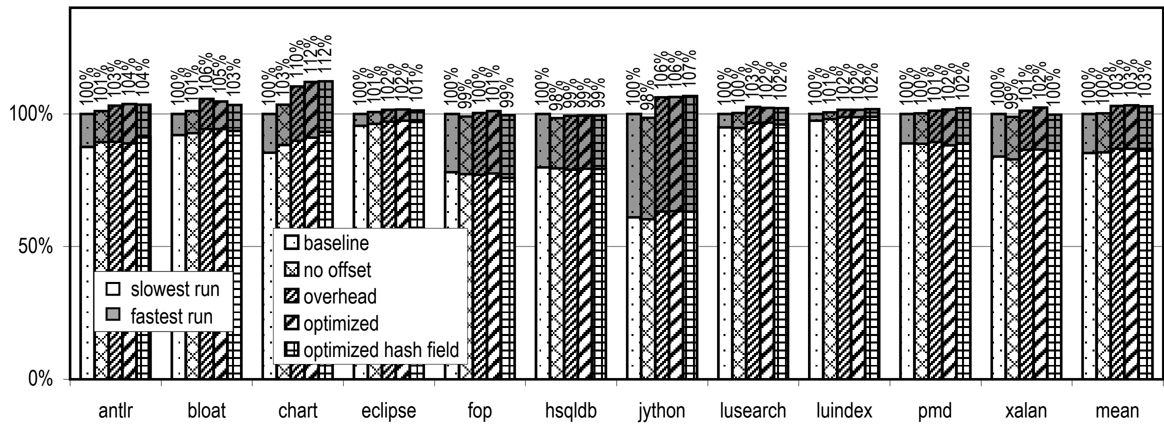


Figure 5.14: DaCapo laptop: performance (higher is better)

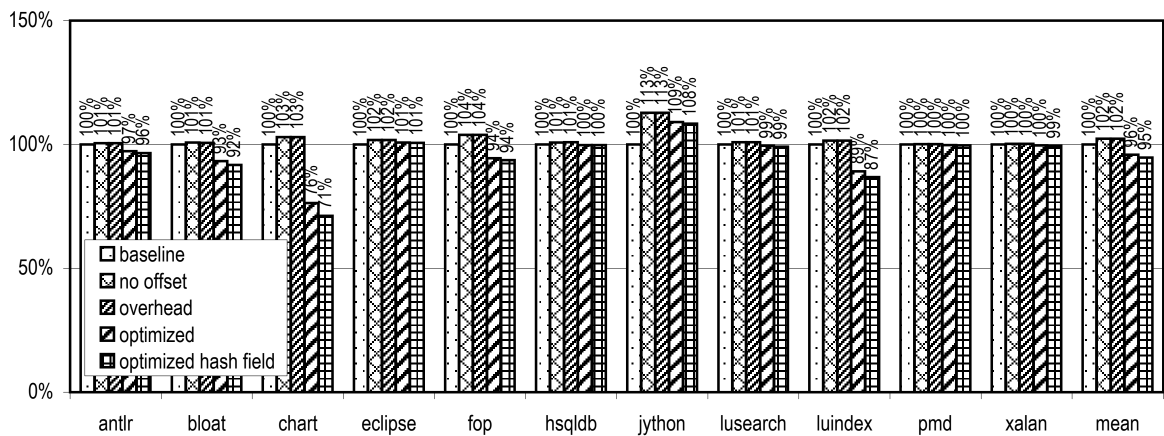


Figure 5.15: DaCapo: number of allocated bytes (lower is better)

the number of garbage collections was measured: the number of full garbage collections is not affected by any of the configurations, and the number of minor garbage collections shows the same reduction as the number of allocated bytes.

When executing this benchmark with the elimination of explicit string copying, no significant different results are measured because the benchmark uses hardly any explicit string copying.

5.4 SPECjvm98

Seven benchmarks that represent typical client applications are included in the SPECjvm98 benchmark suite. Each benchmark is executed multiple times in the same JVM, until the execution time converges. Similar to the DaCapo benchmark, the slowest and the fastest run as well as the geometric mean over all benchmarks are reported.

Figure 5.16 and Figure 5.17 show the performance results for the workstation and the laptop. The slowest and the fastest runs for a benchmark are shown on top of each other. The light bar refers to the slowest run, the dark bar to the fastest run. The string optimization has only a positive impact on the string intensive benchmarks. The performance of the other benchmarks is neither improved nor degraded. Especially the benchmark db, which is string intensive, profits from the string optimization. This benchmark shows a far larger speedup on the workstation.

The configuration "slow hash field" shows a significant negative impact on the performance of the benchmark jack. This configuration implements the removal of the field `hashcode` by invoking the native method `Object.hashCode()`. The invocation of a native method has a small overhead in comparison to the invocation of a Java method. Because the benchmark jack runs only a very short time and computes lots of string hash codes, the summarized overhead reduces the performance significantly. Furthermore, no optimizations are applied to native methods and they are not inlined.

Figure 5.18 shows the number of allocated bytes for the SPECjvm98 benchmark suite. The benchmarks javac, jack, and db show a reduction of the memory usage. Because the number of allocated bytes decreases only slightly for the db benchmark, the main speedup results from the reduced number of field accesses and the better cache behavior. The "no offset" and "overhead" configurations need to allocate significantly more memory than the baseline for the jack benchmark. This indicates that this benchmark uses character array sharing. However, the optimized configurations still need to allocate less memory

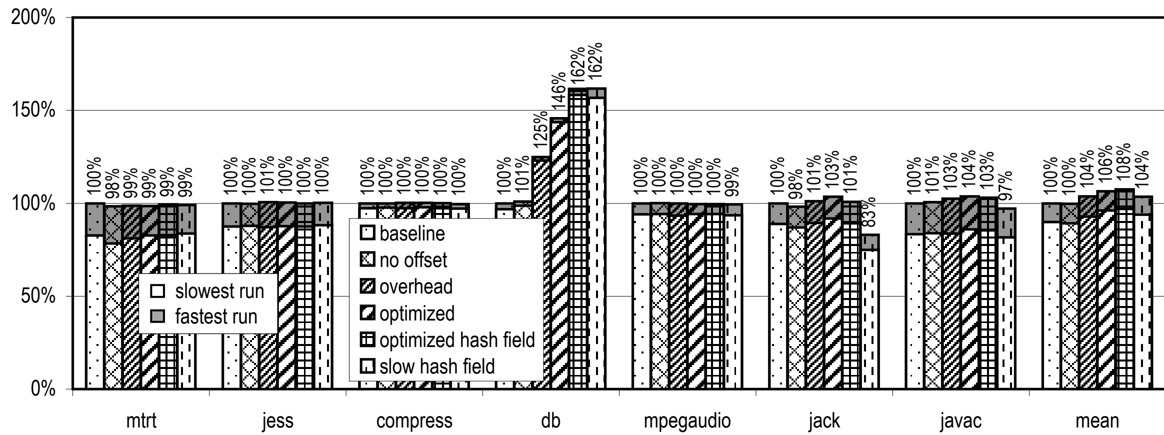


Figure 5.16: SPECjvm98 workstation: performance (higher is better)

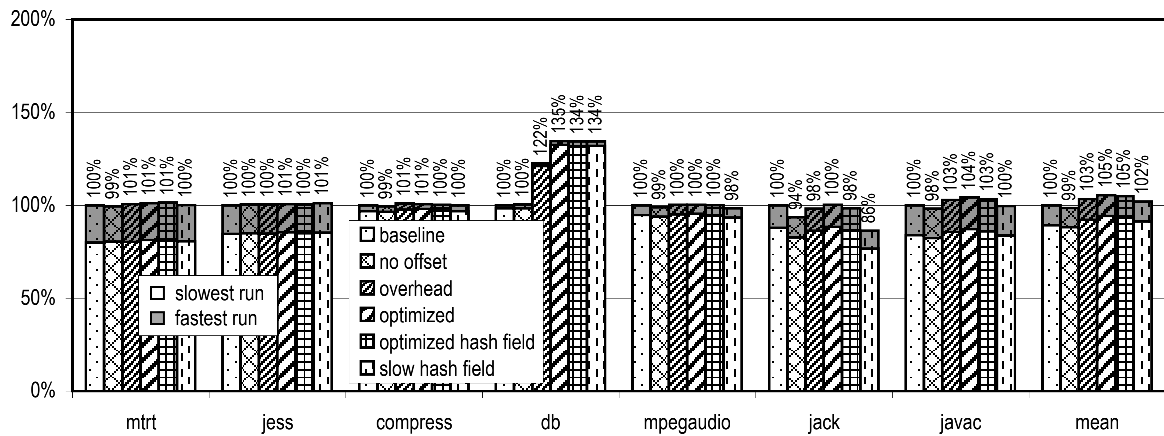


Figure 5.17: SPECjvm98 laptop: performance (higher is better)

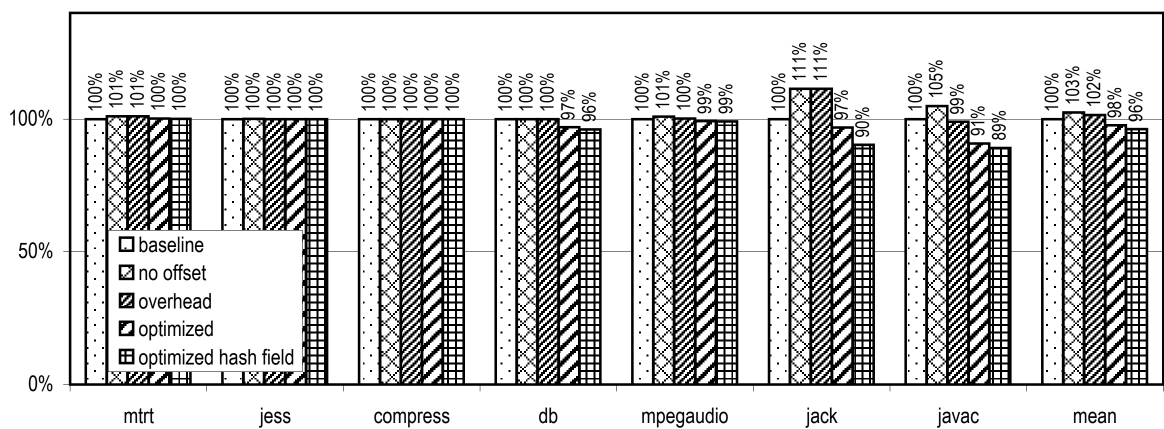


Figure 5.18: SPECjvm98: number of allocated bytes (lower is better)

than the baseline. Furthermore, the number of garbage collections was measured: the number of full garbage collections is not affected by any of the configurations, and the number of minor garbage collections shows the same reduction as the number of allocated bytes. In general, the optimized configurations reduce the memory usage and improve the performance without a significant negative effect on any of the benchmarks. An additional performance measurement with a heap size of 32 MB was performed on the laptop. However, no significant difference to the presented results was measured.

When executing this benchmark suite with the elimination of explicit string copying, nearly no significant difference to the results above is measured because it does not use much explicit string copying. Only the performance of the benchmark javac improves slightly.

Chapter 6

Related Work

Boldi et al. implemented a `MutableString` class that combines the advantages of the classes `String` and `StringBuffer` [4]. 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 exactly to 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 the optimization presented in this thesis, 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 [30]. 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 [31], 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 the optimization presented in this thesis 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 [1]. These techniques include field reduction and the elimination of

unread or constant fields. The value range for each field is analyzed to replace the datatype 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. The approach presented in this thesis optimizes only string objects, but always removes the field `offset` and merges the string's character array with the string object to save additional 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 [7]. 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 the optimization presented in this thesis, 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 [8]. 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. The optimization presented in this thesis 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++ [10]. 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%. The optimization presented in this thesis is performed at run time in a virtual machine and only has a negligible analysis overhead.

Wimmer et al. implemented object inlining for the Java HotSpot™ VM [33, 34]. To determine fields worth inlining, read barriers are used. The just-in-time 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, also the garbage collector was 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 used by multiple string objects. The mean peak performance of the SPECjvm98 benchmark suite is improved by up to 51% (9% on average).

BEA Systems, Inc. has a patent pending to address the inefficiency of `StringBuilder` and `StringBuffer` operations [19]. 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 `toString()` method is invoked, the characters are copied another time to the resulting string object. Therefore, it is more efficient to store the references to the 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 the optimization presented in this thesis is advantageous to the usage of string objects in general.

Zilles implemented *accordion arrays* for Java to reduce the memory usage of character arrays [35]. 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 an 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. The optimization presented in this thesis does not compress any characters, but merges 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% [24]. 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 [25]. 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. The presented optimization in this thesis improves the locality for string objects by merging strings 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 [6]. 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%. The optimization presented in this thesis also uses new bytecodes. However, they are required for the implementation of the optimization and are not mainly used to improve the performance.

Chapter 7

Conclusions

The presented string optimization merges the string and its character array into a single object. For this purpose, new bytecodes are introduced and the constructors of the class `String` are replaced by factory methods. Therefore, all string allocations must be rewritten at run time. The Java HotSpot^{TM} VM automatically performs the optimization at run time so that no actions on part of the programmer are required. The evaluation shows the high potential of the optimization. It has a significant positive impact on the performance and reduces the memory usage for string intensive benchmarks. Furthermore, the garbage collection time is reduced because the garbage collector must handle only one object per string.

7.1 Problems and Further Improvements

Under the scope of this master's thesis, a prototype of the string optimization was implemented to show the potential of the optimization. Most problems encountered during the implementation could be solved. However, some not as crucial problems were not solved for the prototype. If this optimization is to be integrated in a product version of the Java HotSpot^{TM} VM, the following remaining problems must be solved:

- **Reflection:** The optimization replaces the `String` constructors with factory methods. Therefore, no constructors can be accessed via reflection. A similar approach as used for the method resolution of the factory methods could be a possible solution: if a specific constructor is requested, the appropriate factory method could be returned instead. For this, some additional changes in the Java HotSpot^{TM} VM would be necessary. However, this solution might cause problems on the Java side because

the static factory method would be returned in a `Constructor` object. Furthermore, it is possible for original strings to access and modify the fields `offset`, `hashcode`, and `value` via reflection. For a fully optimized string object, none of these fields can be accessed. However, this should not cause severe problems because all fields are declared as private. Code that accesses private fields via reflection should be aware of possible implementation changes.

- **Exception handling:** Exceptions that are thrown 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. Although the factory method's code could be placed in the constructor, constructors are not allowed to return any values. It would still be possible to ignore this restriction and to return the initialized string object anyway, but several parts of the Java HotSpot™ VM rely on the correct return type information.
- **JNI:** Although nearly all problems with the JNI were solved by modifying the method resolution, one problem could not be addressed: 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.
- **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 can be safely reorganized or deleted. These involves more complex operations than the currently implemented heuristic and might introduce a higher overhead. However, the overhead should still be negligible in comparison to the total execution time.
- **Other architectures and the server compiler:** The prototype is only implemented for the IA-32 architecture and the client compiler. It is unlikely that new problems occur when porting the optimization to other architectures. However, adding it to server compiler is some effort because of the changes to the layout helper. Additionally, the server compiler uses several intrinsics to optimize methods of the class `String`. These methods might require additional changes.

7.2 Outlook

Although this prototype already achieves a good speedup, some further optimizations are possible. The optimization ideas discussed in this section differ greatly in their complexity.

- **Helper methods:** Some methods of the class `String` pass the internal character array to a helper method of another class. Optimized strings do not store the string characters any longer in a real character array, as described in Chapter 4.1. Therefore, the characters must be copied to a temporary character array that can be passed to such methods. This could be optimized by overloading the helper method so that a string object can be passed as an argument. However, the characters could then only be accessed via a method, which might still cost some performance.
- **Layout helper:** The original layout helper is only used for arrays and instances. For all other objects, the virtual method `size()` is invoked for the size calculation. The new layout helper, described in Chapter 4.8, could also be used for nearly all VM internal data structures. Only for `symbolOop` objects, it is not possible to use the current implementation because these objects store their element count as a *short* instead of an *integer*.
- **Object transformation:** With the removal of the field `hashcode`, string objects have the same size and memory layout as character arrays. This allows a simple object transformation from character arrays to string objects and vice versa. If a string is initialized with a character array that is no longer needed, the character array could be transformed into the string object. It would also be possible to implement a typecast from one type to the other. However, a programmer could destroy the immutability of string objects with such a typecast.
- **Lazy array initialization:** The characters of optimized string objects do not need to be initialized with a default value because the factory method initializes all characters anyway. A similar approach could be used for array allocation. Currently, all array elements are initialized with a default value, even if later instructions overwrite the default content. An optimization for the JIT compiler could be implemented to detect arrays where the initialization with the default values can be safely omitted.
- **Stringcopy:** Since it is no longer possible to use the method `System.arraycopy()` for copying characters from or to an optimized string, the three methods `String.stringToStringCopy()`, `String.stringToCharCopy()` and `String.charToStringCopy()` were introduced (see Chapter 4.10). These methods share most of its code with the very general method `System.arraycopy()`. A speedup might be achieved by implementing an intrinsic method that is specialized on copying characters.

List of Figures

2.1 Execution of a method . . . 5
2.2 Class loading . . . 6
2.3 Constant pool example . . . 7
2.4 Bytecodes used for allocation . . . 12
2.5 Heap with a string object and 8 byte alignment . . . 13
2.6 Java object header on 32-bit architectures . . . 14
2.7 Possible bit patterns for the *mark word* . . . 15

3.1 Layout of Java strings . . . 22
3.2 Layout of optimized strings . . . 23
3.3 Cache behavior of strings . . . 25
3.4 Reduction of memory usage for the optimized string class (higher is better) 26
3.5 Method `String.substring()` . . . 27

4.1 New bytecodes for accessing a character of an optimized string . . . 34
4.2 Stack before and after the execution of the bytecode **`newstring`** . . . 36
4.3 Bytecode rewriter component . . . 37
4.4 Simplified overview of the used *oops* in the HotSpot™ VM . . . 42
4.5 Example of all required *oops* for a `StringBuffer` object . . . 43
4.6 Bit patterns of the original layout helper . . . 44
4.7 Bit patterns of the new layout helper . . . 46
4.8 Subset of the HIR class hierarchy . . . 49
4.9 Subset of the LIR class hierarchy . . . 50
4.10 Explicit string copying factory method . . . 52

5.1 SPECjbb2005: benchmark results . . . 56
5.2 SPECjbb2005 workstation: performance (higher is better) . . . 56
5.3 SPECjbb2005 workstation: garbage collection time (lower is better) . . . 56
5.4 SPECjbb2005 laptop: performance (higher is better) . . . 57
5.5 SPECjbb2005 laptop: garbage collection time (lower is better) . . . 57
5.6 SPECjbb2005 laptop: average used memory after a full garbage collection (lower is better) . . . 57

5.7 SPECjbb2005: benchmark results 60
5.8 SPECjbb2005 workstation: performance (higher is better) 60
5.9 SPECjbb2005 workstation: garbage collection time (lower is better) 60
5.10 SPECjbb2005 laptop: performance (higher is better) 61
5.11 SPECjbb2005 laptop: garbage collection time (lower is better) 61
5.12 SPECjbb2005 laptop: average used memory after a full garbage collection (lower is better) 61
5.13 DaCapo workstation: performance (higher is better) 63
5.14 DaCapo laptop: performance (higher is better) 63
5.15 DaCapo: number of allocated bytes (lower is better) 63
5.16 SPECjvm98 workstation: performance (higher is better) 65
5.17 SPECjvm98 laptop: performance (higher is better) 65
5.18 SPECjvm98: number of allocated bytes (lower is better) 65

List of Listings

2.1 Interpreter loop 8
2.2 Example usage of the Java Native Interface 17
2.3 Example usage of reflection 17
2.4 Example usage of assertions 18
2.5 Example for impossible cases 19
2.6 Example for self-verification 20

4.1 Method `String.charAt()` 29
4.2 Method `String.substring()` 29
4.3 Method `String.getBytes()` 30
4.4 Java HotSpot^{TM} VM method `utf8_length` 31
4.5 Method `String.hashCode()` 33
4.6 Bytecodes of the method `String.charAt()` 35
4.7 A factory method 36
4.8 Bytecode rewriting heuristic 38
4.9 Example for bytecode rewriting 39
4.10 Factory method compiled with the modified version of javac 41
4.11 The original fast object size computation 45
4.12 The new fast object size computation 46
4.13 Interpreter code patterns for `caload` and `castore` 47
4.14 Interpreter code pattern for `newstring` 48
4.15 `String.hashCode()` compiler intrinsic 51

Bibliography

[1] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 59–68. ACM Press, 2003.

[2] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[3] 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, and 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*, pages 169–190. ACM Press, 2006.

[4] Paolo Boldi and Sebastiano Vigna. Mutable strings in Java: Design, implementation and lightweight text-search algorithms. *Science of Computer Programming*, 54(1):3–23, 2005.

[5] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley, 1998.

[6] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizations for a Java Interpreter Using Instruction Set Enhancement. Technical report, Department of Computer Science, University of Dublin, Trinity College, 2005.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 282–301. ACM Press, 2003.

[8] Guangyu Chen, Mahmut Kandemir, and Mary 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*, pages 68–78. ACM Press, 2005.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control depen-

dence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[10] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357. ACM Press, 2000.

[11] Bruce Eckel and Chuck Allison. *Thinking in C++, Volume 2: Practical Programming*. Prentice Hall, 2003.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, 3rd edition, 2005.

[13] Robert Griesemer and Srdjan Mitrovic. A compiler for the Java HotSpot™ virtual machine. In Lázló Böszörményi, Jürg Gutknecht, and Gustav Pomberger, editors, *The School of Niklaus Wirth: The Art of Simplicity*, pages 133–152. dpunkt.verlag, 2000.

[14] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[15] Intel Corporation. *The IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. `http://www.intel.com/products/processor/manuals/`.

[16] Richard Jones and Rafael Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.

[17] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 111–120. ACM Press, 2005.

[18] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7, 2008.

[19] Staffan Larsen. What's Hot in BEA JRockit, 2007. BEA Systems, Inc., `http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2171.pdf`.

[20] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley, 1999.

[21] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[22] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12. USENIX, 2001.

[23] Kenneth Russell and David 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*, pages 263–272. ACM Press, 2006.

[24] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–306. ACM Press, 2002.

[25] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal 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*, pages 13–25. ACM Press, 2002.

[26] Standard Performance Evaluation Corporation. *The SPECjvm98 Benchmarks*, 1998. `http://www.spec.org/jvm98/`.

[27] Standard Performance Evaluation Corporation. *The SPECjbb2005 Benchmark*, 2005. `http://www.spec.org/jbb2005/`.

[28] Sun Microsystems. *Memory Management in the Java HotSpot™ Virtual Machine*, 2007. `http://java.sun.com/j2se/reference/whitepapers/`.

[29] Sun Microsystems, Inc. *Java Platform, Standard Edition 7 Source Snapshot Releases*, 2007. `http://download.java.net/jdk7/`.

[30] Ye Henry Tian. String concatenation optimization on Java bytecode. In *Proceedings of the Conference on Programming Languages and Compilers*, pages 945–951. CSREA Press, 2006.

[31] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot – A Java optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135. IBM Press, 1999.

[32] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141. ACM Press, 2005.

[33] Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object inlining in the Java HotSpot™ virtual machine. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 12–21. ACM Press, 2007.

[34] Christian Wimmer and Hanspeter Mössenböck. Automatic array inlining in Java virtual machines. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 14–23. ACM Press, 2008.

[35] Craig Zilles. Accordion arrays. In *Proceedings of the International Symposium on Memory Management*, pages 55–66. ACM Press, 2007.

Curriculum Vitae

Personal data	
Name:	Christian Georg Häubl
Date of birth:	December 25, 1984
Family status:	single
Citizenship:	Austria
Parents:	Maria Häubl
	Dipl.-Ing. Dr. Georg Häubl
Siblings:	Dipl.-Ing. Martin Häubl

Education	
1991 – 1995	Primary school
1995 – 2003	Secondary school
June 2003	Final exam passed with distinction
2003 – 2004	Alternative civilian service
2004 – 2007	Bachelor study of computer science
	Johannes Kepler University Linz
April 2007	Passed bachelor study of computer science
since April 2007	Master study of computer science
since October 2007	Master study of networks and security