Optimized Strings for the Java HotSpot™ Virtual Machine

Christian Häubl Christian Wimmer Hanspeter Mössenböck
Institute for System Software
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz
Linz, Austria
{haeubl, wimmer, moessenboeck}@ssw.jku.at

ABSTRACT

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

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

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

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Code generation*

General Terms

Languages, Performance

Keywords

Java, string, optimization, performance

PPPJ 2008, September 9–11, 2008, Modena, Italy.
http://doi.acm.org/10.1145/1411732.1411747

1. INTRODUCTION

Strings are one of the essential data structures used in nearly all programs. Therefore, string optimizations have a large positive effect on many applications. Java supports string handling at the language level [8]. However, all string operations are compiled to normal method calls of the classes `String` and `StringBuilder` in the Java bytecodes [11].

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

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

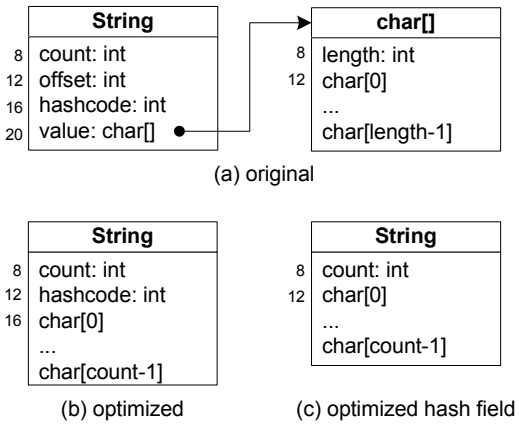

Figure 1: Object layout of strings

Although string objects can share their character arrays, this is not the common case. We measured the percentage of strings that do not use their full character array: 0.05% for the SPECjbb2005 [15] benchmark, 5% for the SPECjvm98 [14] benchmarks, and 14% for the DaCapo [2] benchmarks. However, a string also shares its character array when it is explicitly copied. Of all string allocations, 19% are explicit string copies for the SPECjbb2005 benchmark and 4% for the SPECjvm98 benchmark. The DaCapo benchmarks allocate hardly any explicit string copies.

Because of these results, we propose different string layouts as shown in Figure 1 (b) and (c). In the first one, we remove the field `offset` and merge the character array with the string object. This precludes the sharing of character arrays between string objects, but has several advantages such as the reduction of memory usage and the elimination of field accesses. The second variant, described in Section 4.5, also removes the field `hashcode` to save another four bytes per string object. The computed hash code is cached in the object header instead. Our paper contributes the following:

- We present two string optimization variants that reduce the memory usage and increase the performance.
- Our approach requires neither actions on the part of the programmer nor any changes outside the Java HotSpot™ VM.
- The evaluation shows the impact of our optimizations on the number of allocated bytes and the performance of several benchmarks.

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

2. SYSTEM OVERVIEW

We build on the early access version b24 of Sun Microsystems' Java HotSpot™ VM, which is part of the upcoming JDK 7 [16]. The VM is available for multiple architectures, however our string optimization is currently only implemented for the IA-32 architecture because platform dependent code is necessary within the interpreter and the just-in-time compiler.

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

- The *client compiler* is optimized for compilation speed and refrains from time consuming optimizations [9, 10]. With this strategy, the application startup time is low while the generated machine code is still reasonably well optimized.
- The *server compiler* makes use of more sophisticated optimizations to produce better code [12]. It is designed for long-running server applications where the initial compilation time is irrelevant, and where a high peak performance is essential.

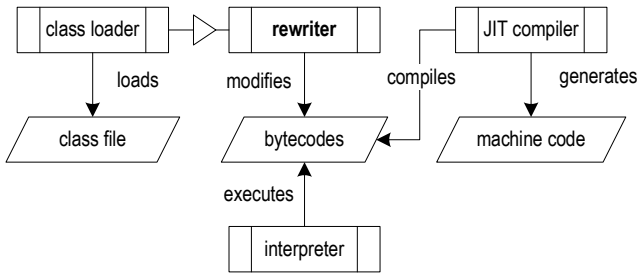

Figure 2: System overview

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

Every Java object has a header of two machine words, i.e. 8 bytes on 32-bit architectures, 16 bytes on 64-bit architectures [13]. 7 bits of the first header word are used for synchronization and during garbage collection. The remaining 25 or 57 bits of the first word store the identity hash code of the object when it is computed via the method `System.identityHashCode()`. The second header word stores a pointer to the class descriptor object that is allocated when a class is loaded. The class descriptor holds the metadata and the method table of the class.

Currently, the optimization is only implemented for the interpreter and the client compiler. The client compiler uses a graph-based high-level intermediate representation (HIR) as well as a low-level intermediate representation (LIR) [10]. The HIR is in static single assignment (SSA) form [6] and is used for global optimizations. The LIR is used for linear scan register allocation and for peephole optimizations.

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

3. ADVANTAGES OF THE OPTIMIZATION

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

- *Elimination of field accesses*: By removing the field `offset`, one field access is saved in almost every string

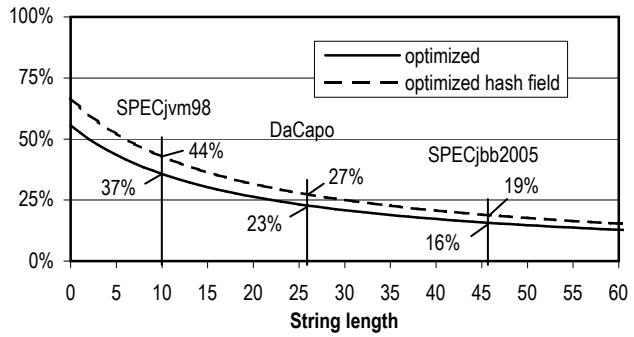

Figure 3: Reduction of memory usage for optimized string objects (higher is better)

operation. By embedding the character array into the string object, the characters can be accessed without dereferencing the array pointer.

- *Reduced memory usage*: Original string objects have a minimum size of 36 bytes. With our optimization, the minimum size is 12 or 16 bytes, depending on the removal of the field `hashcode`. Saving up to 24 bytes per string object results in the reduced memory usage shown in Figure 3, which depends on the string length. For example, for strings of length 10 our optimized string handling saves 37% of string memory, or 44% if also the field `hashcode` is eliminated. The figure also contains the average string lengths for the different benchmark suites. Because the memory usage is reduced, fewer garbage collections are necessary.

- *Faster garbage collection*: An original string is composed of a string object and a character array that both must be processed by the garbage collector. The optimized string is a single object and thus reduces the garbage collection time.

- *Better cache behavior*: The two parts of an original string can be spread across the heap. Both parts are always accessed together, which results in a bad cache behavior. The optimized string is always one unit that possibly fits into a single cache line.

4. IMPLEMENTATION

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

To introduce the new bytecodes, it is necessary to transform the object code of the `String` class. Although this could be done at run time, it would be complicated. Therefore, we modify the Java compiler (javac) and use it to compile the class `String`. This results in an optimized class, which is created once, and is used by the VM if the string optimization is enabled. The modified version of javac is not used for compiling any other source code.

Additionally, methods that allocate string objects must be transformed. This must be done at run time because it affects application classes whose source code is not available. The transformation details are explained in Section 4.4.

Introducing new bytecodes for optimized operations inside the VM is a common pattern. These bytecodes use numbers that are unused according to the specification [11]. Because it is only necessary to handle the new bytecode instructions in the interpreter and to support them in the just-in-time compiler, the impact on the overall VM structure is low.

4.1 Removing the Offset Field

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

- In most cases, it is possible to just remove the accesses to the field `offset`, or to replace them with the constant 0. An example for this case is the method `String.charAt()`.

- Sometimes, the field `offset` is used to implement an optimization such as the sharing of character arrays between string objects. This kind of optimization is no longer possible and it is necessary to create a copy of the characters leading to a certain overhead. An example for this is the method `String.substring()`.

- In uncommon cases, the string-internal character array is passed to a helper method of another class. The optimized string characters cannot be passed as a character array to a method anymore. It would be necessary to pass the string object itself. To make this possible, the receiving method would have to be overloaded to allow a string object instead of a character array as an argument. We decided to keep the number of changes to a minimum and did not change or add any methods outside of the class `String`. Instead, we copy the string characters to a temporary character array which is then passed as an argument to such methods. This is expensive, but could be easily optimized in the future.

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

4.2 Character Access

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

- `scload`: This bytecode loads a string character and is similar to the bytecode `caload` used for loading a character from a character array. `scload` expects two operands on the stack: a reference to the string object and the index of the accessed character.

- `scstore`: This bytecode stores a string character and is similar to the bytecode `castore` used for storing a character in a character array. Compared to the `scload` bytecode, one additional operand is expected on the stack: the character which is to be stored at the specified index of the given string.

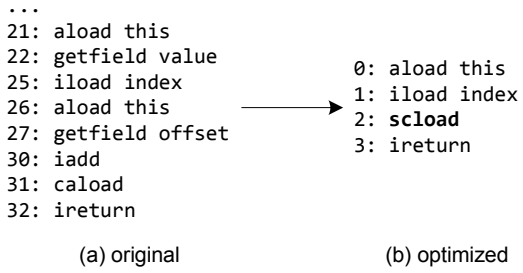

Figure 4: Bytecodes of the String.charAt method

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

- As illustrated in Figure 1, the offset of the first character of an optimized string is different from the offset of the first element of a character array. Furthermore, the offset depends on the optimization level: it is 16 if the field `hashcode` is preserved, and 12 if it is removed.
- Each time a character array is indexed, a bounds check is performed. If the index is out of the valid bounds, an `ArrayIndexOutOfBoundsException` is thrown. The optimized string object needs a bounds check too, but within the `String` class it is a convention to throw a `StringIndexOutOfBoundsException` if the check fails.

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

4.3 String Allocation

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

We solve this problem by replacing the string constructors with static factory methods that combine the object allocation and initialization. These factory methods have the same arguments as the constructors. They calculate the length of the resulting string, allocate the optimized string object, and initialize it. One of these factory methods is presented in Figure 5. The operator *newstring*, used in this method, is compiled to a new bytecode for string allocation by the modified javac. The details of this bytecode are explained below.

Figure 6 (a) presents a simple Java method. The method allocates and returns an optimized string object that is initialized using a character array. The bytecodes of the unoptimized version are shown in Figure 6 (b). Because the size of the original string object is statically known, it can be allocated. In the next step, the reference to the newly created original string object is duplicated on the operand stack. This is necessary because the reference is needed both for the invocation of the constructor and the method return. After that, the constructor's parameter is pushed onto the operand stack and the constructor is invoked. The constructor computes the length of the string's character array, allocates it, and initializes it with the characters of the array `ch`.

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

Figure 5: Factory method for string allocation

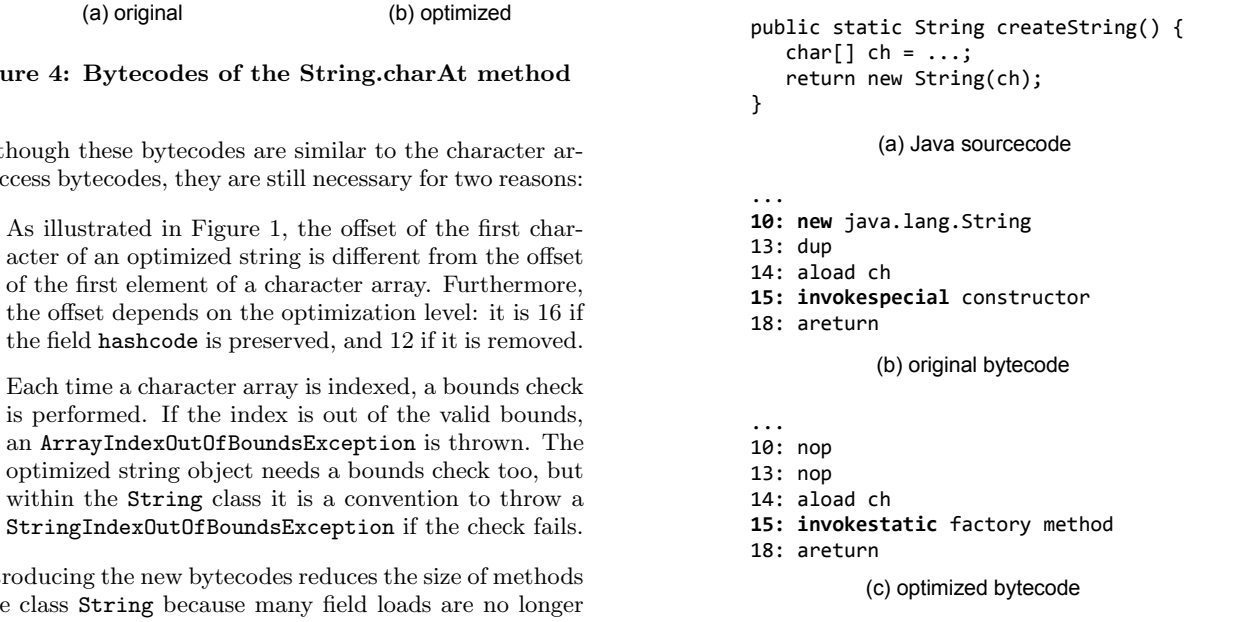

Figure 6: Example for string allocation

The previously introduced *rewriter* component transforms the original bytecodes to the optimized version shown in Figure 6 (c) (the details about the bytecode rewriting algorithm are presented in Section 4.4). The transformation happens whenever a method is loaded that allocates string objects. The string allocation and the subsequent reference duplication are replaced with no operation (`nop`) bytecodes. We do not remove these bytecodes completely because this would change the bytecode indices and thus would have side effects on all jumps within the method. If the method is compiled, the `nop` bytecodes are ignored anyway.

Figure 7 shows the bytecodes available for allocating objects and arrays. For the allocation of an original string object, the bytecode `new` is used as illustrated in Figure 6 (b). This bytecode can only be used if an object with a statically known size is to be allocated, which applies to all Java objects except arrays. The only operand is an index to a class in the constant pool. When the bytecode is executed, the index is used to fetch a class descriptor that contains the

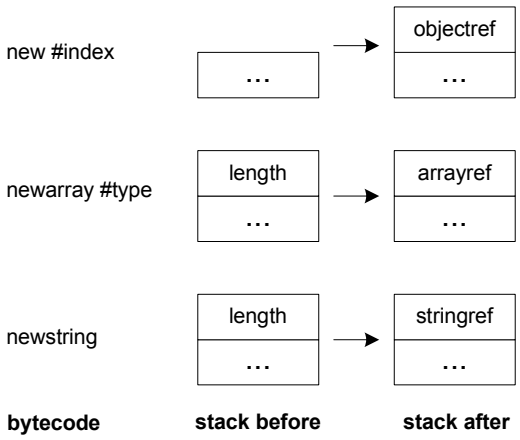

Figure 7: Allocation bytecodes

object size. Knowing the size, an object of this class can be allocated.

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

Optimized string objects have a variable length like arrays, but also fields like objects, so neither of the two previous bytecodes can be used. Therefore, we introduce the bytecode `newstring` that is similar to the bytecode `newarray` and expects the string length on the stack. A bytecode operand, like the element type, is not necessary because the VM knows that a string object only contains characters. Furthermore, the number of fields of a string is fixed and statically known. With this knowledge, the total string size is calculated and the allocation is performed. This bytecode is exclusively used to implement the allocation of optimized string objects within the factory methods (the emphasized part in Figure 5).

4.4 Bytecode Rewriting

The *rewriter* component transforms the original, unoptimized bytecodes to the optimized ones. This is necessary for all methods that allocate string objects. Whenever a class has been loaded, the *rewriter* checks if a method of this class allocates string objects. If this is the case, the *rewriter* transforms the bytecodes in three steps, as illustrated in Figure 8:

1. The allocation of the string object, i.e. the bytecode `new`, is removed by replacing it with `nop` bytecodes. The bytecode `new` would push a string reference on the stack which does not happen anymore because of the removal.

2. The bytecode rewriting process is difficult because of stack management instructions like `dup` or `pop` that would use the no longer existing string reference. Each of these management instructions must be modified or removed. Furthermore, some of the subsequent bytecodes might have to be rearranged. Our current implementation is prototypical in that it handles only the most common stack management instructions. Yet, it is complete enough for running nearly all Java programs including the various benchmarks presented in Section 5. Only special bytecode-optimized programs which make use of more sophisticated but rarely used stack management instructions like `dup_x1` might cause problems and are currently not supported (i.e. they are reported as errors).

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

Figure 8: Bytecode rewriting heuristic

3. In the last step of the rewriting process, the constructor invocation is replaced with the invocation of a factory method. Because the factory method has the same arguments as the constructor, no change to the operand stack handling is necessary. This rewriting step can be implemented in two different ways:

   - The constant pool index of the invoked method could be rewritten. This would make it necessary to add the name of the factory method to the constant pool.
   - The method resolution within the VM could be modified. Each time a string constructor must be resolved, the factory method is returned instead. This also works for applications which use the Java Native Interface (JNI) and could not be rewritten otherwise. Therefore, we implemented this approach.

4.5 Removing the Hashcode Field

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

Every Java object has a header of two machine words. If the identity hash code of an object is calculated via the method `System.identityHashCode()`, the result is cached

in the header (and is truncated to 25 bits on 32-bit architectures). This caching is necessary because although the identity hash code is a random number that does not depend on the object's value, it must not change during the object's lifetime.

Although the identity hash code is preferably unique for every object, it is not guaranteed to be so. Therefore, it should not have any side effects if the identity hash code of strings is equal to the string hash code. So, we use the object header for caching the string hash code. As long as `String.hashCode()` is executed by the interpreter, the hash code is not cached and has to be calculated upon each method invocation. This is necessary because the Java object header cannot be accessed via a bytecode. When the method `String.hashCode()` is passed to the just-in-time compiler, it is not compiled but an intrinsic method is used that calculates the hash code once and caches its value in the object header. Only when the object is locked and the header word is used to point to a locking data structure, we do not store the hash code because this code path would be more expensive than recomputing the hash code. This is a rare case because strings are normally not used for synchronization.

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

4.6 Further Adjustments

Some existing optimizations are voided by the new optimized class `String`. The method `System.arraycopy()` can no longer be used to create a copy of the string characters because they are not stored as a real array anymore. `System.arraycopy()` is faster than a loop in Java code because some bounds and type checks can be omitted. Therefore, we use specialized versions of `System.arraycopy()` for optimized string objects. These methods copy a range of characters between two strings or between a string and a character array. In the just-in-time compiler, these new methods are handled as intrinsic methods and share nearly the whole code with the method `System.arraycopy()`.

Because of replacing the string constructors with factory methods, no constructors can be accessed via reflection. This could be solved by returning the appropriate factory method if a specific constructor is requested. This would require some additional changes in the Java HotSpot™ VM, which are not implemented yet.

5. EVALUATION

Our string optimization is integrated into Sun Microsystems' Java HotSpot™ VM, using the early access version b24 of the upcoming JDK 7. The benchmarking system has the following configuration: an Intel Core2 Quad processor with 4 cores running at 2.4 GHz, 2 * 2 MB L2 cache, 2 GB main memory, and with Windows XP Professional as the operating system. For measuring the performance and the number of allocated bytes, we use the SPECjvm98 [14], SPECjbb2005 [15], and the DaCapo [2] benchmarks. We present the results of four different configurations:

- Our baseline configuration is the unmodified Java 7 build b24.
- In the "optimized" configuration, all optimizations described in this paper except the removal of the field `hashcode` are performed.
- Our optimization has two benefits: a reduced memory consumption because of the smaller string objects, and a reduced number of memory accesses because of the eliminated fields. The "overhead" configuration increases the size of the optimized string objects artificially to the size of the original strings by adding a padding of 20 bytes per string. If a benchmark does not invoke any methods that use character array sharing for the original string, this configuration allocates exactly the same number of bytes as the baseline. Any additional memory overhead (e.g. in Figure 14) is a result of the missing character array sharing. Therefore, this configuration indicates how much character array sharing is used in a benchmark. The performance loss in comparision with the "optimized" configuration shows the impact of the reduced memory usage on the performance.
- The "optimized hash field" configuration uses all optimizations described in this paper including the optimization of the field `hashcode`. We use a 32-bit architecture for benchmarking and therefore the hash code is truncated to 25 bits. Because the SPECjbb2005 benchmark compares the hash code of a string object to a hardcoded value during the startup, the benchmark had to be modified slightly.

5.1 SPECjbb2005

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

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

Figure 9 illustrates the SPECjbb2005 performance and the average amount of used memory after a full garbage collection. The used memory after a full garbage collection serves as an approximation of the application's minimum heap size. Both numbers are significantly improved by our string optimization.

The SPECjbb2005 benchmark always runs 240 seconds for a specific number of warehouses. Therefore, the total number of allocated bytes depends on the performance, i.e. on how many transactions can be run in this time frame. To measure the number of allocated bytes independently of

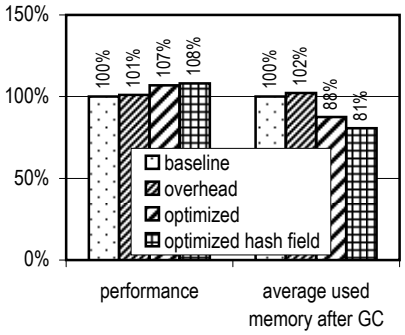

Figure 9: SPECjbb2005: performance and memory usage

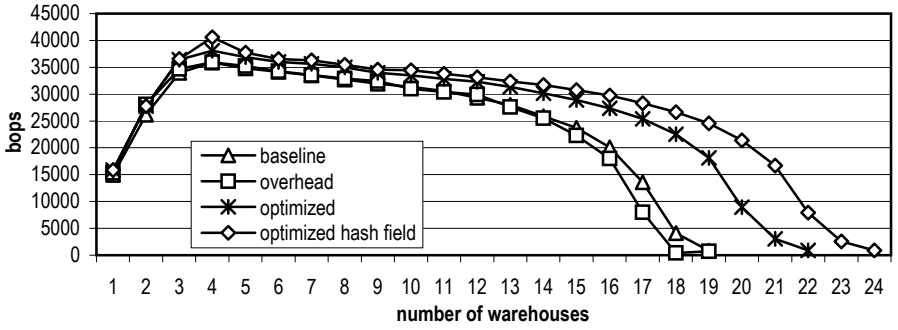

Figure 10: SPECjbb2005: performance for various numbers of warehouses (higher is better)

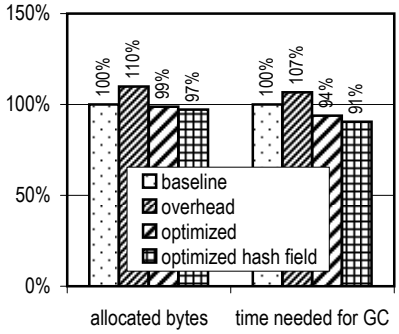

Figure 11: SPECjbb2005: memory usage for a fixed number of transactions (lower is better)

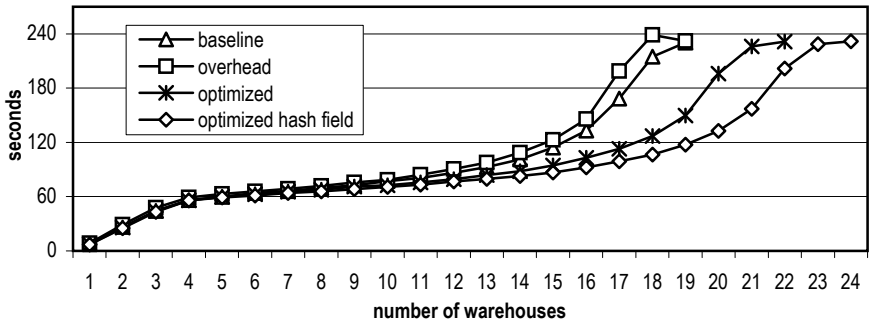

Figure 12: SPECjbb2005: garbage collection time for various numbers of warehouses (lower is better)

the performance, we used a slightly modified version of the SPECjbb2005 benchmark that executes a fixed number of transactions on four warehouses. For the optimized configurations, the number of allocated bytes is reduced as shown in Figure 11. Furthermore, the optimizations also reduce the time necessary for garbage collection. The configuration "overhead" allocates more bytes than the the baseline because original strings use character array sharing for explicit string copying.

To further evaluate the impact of the reduced memory consumption, we executed each configuration with a heap size of 512 MB, up to the number of warehouses where an `OutOfMemoryException` was thrown. Figure 10 shows the performance for various numbers of warehouses. The performance increases up to 4 warehouses because each warehouse uses its own thread and the benchmarking system has 4 cores. With a higher number of warehouses, the thread overhead and the memory usage increases, so that the performance decreases. Because of the reduction of memory usage, it is possible to execute the SPECjbb2005 benchmark with 24 instead of 19 warehouses.

Figure 12 shows the average garbage collection time for runs with various numbers of warehouses. The configurations that reduce the memory usage also spend less time in the garbage collector. The memory-resident database uses a smaller part of the heap and therefore more memory is available for the actual execution, which then needs fewer garbage collections. There is a clear correlation between Figure 10 and Figure 12: the performance decreases as the time for garbage collection increases.

5.2 DaCapo

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

The performance for the benchmarks in the DaCapo suite is presented in Figure 13. In this diagram, the slowest and the fastest runs for each benchmark are shown on top of each other. Both runs are shown relative to the fastest run of the baseline. The light bars refer to the slowest runs, the dark bars to the fastest runs.

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

The number of allocated bytes for each benchmark is presented in Figure 14. Again, the chart benchmark profits

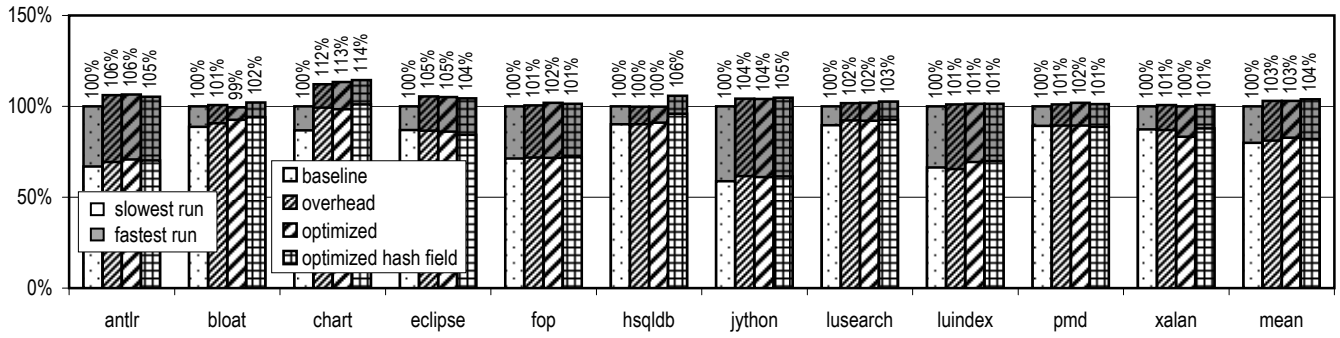

Figure 13: DaCapo: performance results (higher is better)

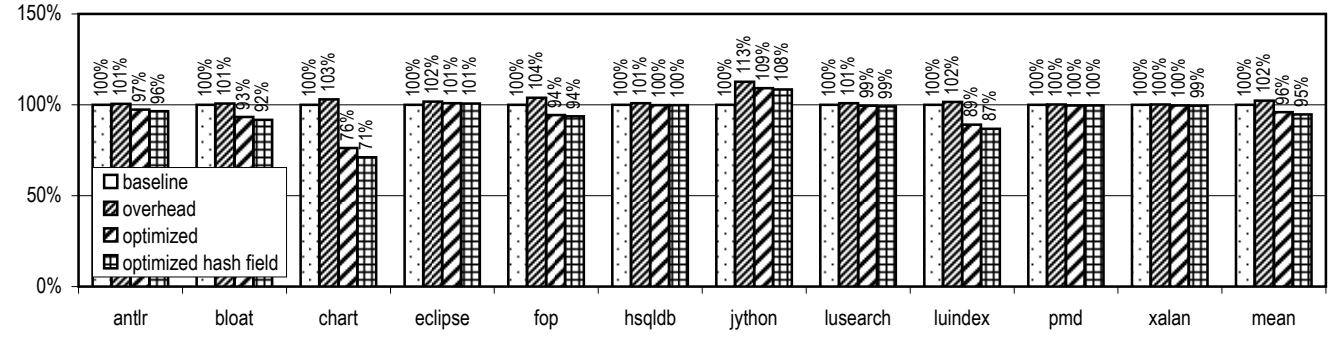

Figure 14: DaCapo: number of allocated bytes (lower is better)

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

5.3 SPECjvm98

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

Figure 15 illustrates the results of the SPECjvm98 benchmark suite. In this diagram, the slowest and the fastest run for each benchmark are shown on top of each other. Both runs are shown relative to the fastest run of the baseline and the light bars refer to the slowest runs, the dark bars to the fastest runs.

Especially the db benchmark, which mainly operates on string objects, profits from the string optimization. The number of allocated bytes decreases only slightly, so the high speedup results from the removal of the field accesses and the better cache behavior. For all benchmarks the performance of the slowest run as well as of the fastest run is greater or equal to the baseline. Furthermore, the number of allocated bytes is smaller or equal to the baseline for all benchmarks. This means that the performance and the number of allocated bytes are optimized without any negative effect on any of the benchmarks. The largest reduction of the number of allocated bytes is measured for the jack benchmark.

5.4 Further Evaluations

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

About 19% of all string allocations in the SPECjbb2005 benchmark are explicit string copies. These copies are allocated with the constructor `String(String)`, which trims the internal character array for original strings. Because optimized strings always use all their characters, explicit copies are useless. However, they cannot be removed easily for two reasons:

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

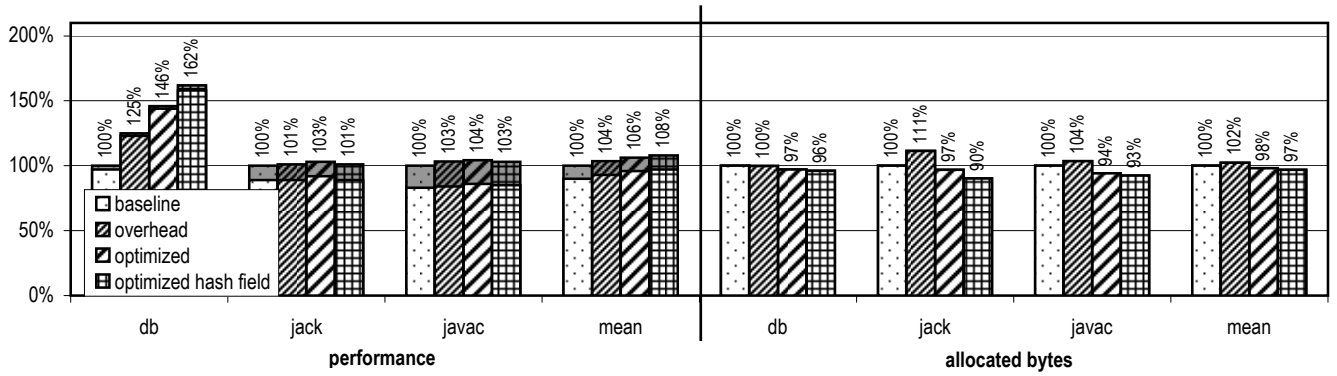

Figure 15: SPECjvm98: performance and allocated bytes for string intensive benchmarks

These cases would have to be detected to safely eliminate explicit string copying. Both cases do not apply to any of the explicit string copies in the SPECjbb2005 benchmark. To measure which performance could be expected from the string optimization if the programmer knows that the allocation of explicit string copies is unnecessary, all explicit string copies were removed for the SPECjbb2005 benchmark. In comparison to the baseline, the "optimized hash field" configuration shows a 18% higher performance and a reduction of the average used memory after a full garbage collection by 20%. Furthermore, the number of allocated bytes is reduced by 11%, and the time necessary for garbage collection is reduced by 30%.

6. RELATED WORK

Boldi et al. implemented a `MutableString` class, which combines the advantages of the classes `String` and `StringBuffer` [3]. A `MutableString` can be in the state "compact" or "loose". Depending on this state, the `MutableString` has the advantages of the class `String` or `StringBuffer`. In contrast to our optimization, existing programs must be changed and recompiled to be able to use the advantages of this newly introduced class. Furthermore, their optimization does not reduce the memory usage.

Tian addressed the performance problem of string concatenations [17]. Each time two string objects are concatenated, a new string object is allocated and all characters are copied. Using the Java bytecode optimization framework Soot [18], a bytecode transformation was implemented. With this transformation, it is possible to improve the performance of string concatenation nearly up to the performance of the class `StringBuilder`. This optimization has to be applied directly to the .class file, while our optimization is performed automatically behind the scenes by the VM and covers more than just 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. Furthermore, static specialization is used to create two different string classes: one string class without an field `offset` (`SmallString`) and one with a field `offset` (`BigString`). If the field `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, some have a positive one. We only optimize string objects, but we always remove the field `offset` and we merge the string's character array with the string object, which saves further memory.

Chen et al. implemented two different approaches to reduce the memory consumption. A heap compression algorithm reduces the minimum heap size of an application [5]. It mainly targets memory constrained environments such as mobile devices and uses a Mark-Compact-Compress-Lazy Allocate garbage collector as its basic component. The approach can use an arbitrary compression algorithm. The evaluation was done with a "zero removal" compression algorithm, and showed that 35% of memory can be saved on average. The compression can be applied to all kinds of Java objects, but has a negative impact on the overall performance.

The second approach exploits frequent field values to reduce the memory usage [4]. It uses the fact that a small number of distinct values appear in a lot of fields. Two object compressions are proposed to reduce the amount of memory for fields. The evaluation with SPECjvm98 shows that the minimum heap size, which allows Java applications to execute without an `OutOfMemoryException`, is reduced by up to 24% (14% on average). The loss of performance is below 2% for most of the cases. Our approach optimizes only string objects but shows a good 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++ [7]. This optimization can merge some referenced objects with the referencing one, which improves the cache behavior and reduces the indirection overhead. The average speedup for the C++ benchmarks is 10% with a maximum of 50%. However, the necessary analysis for object inlining takes up to half of the compilation time. In contrast to our optimization, object inlining is not limited to string objects, but our optimization is performed at run time in a virtual machine and has only a negligible analysis overhead.

Wimmer et al. implemented object inlining for the Java HotSpot^{TM} VM [19, 20]. The optimization is performed automatically at run time, but cannot optimize strings because the character array can be shared between multiple string objects. The mean peak performance of the SPECjvm98 benchmark suite is improved by 9% (with a maximum of 51%).

7. CONCLUSIONS

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

8. REFERENCES

[1] C. S. Ananian and M. 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] 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.

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

[4] G. Chen, M. Kandemir, and M. J. Irwin. Exploiting frequent field values in Java objects for reducing heap memory requirements. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 68–78. ACM Press, 2005.

[5] 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.

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

[7] J. Dolby and A. 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.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. Addison-Wesley, 3rd edition, 2005.

[9] R. Griesemer and S. Mitrovic. A compiler for the Java HotSpot™ virtual machine. In L. Böszörményi, J. Gutknecht, and G. Pomberger, editors, *The School of Niklaus Wirth: The Art of Simplicity*, pages 133–152. dpunkt.verlag, 2000.

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

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

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

[13] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 263–272. ACM Press, 2006.

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

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

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

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

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. 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.

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

[20] C. Wimmer and H. 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.